# TI-89 / TI-92 Plus Tip List 10.0

Compiled by Doug Burkett, dburkett@infinet.com

Created 12 aug 99, revised  20 July 2002

Dedicated to my wife,
Janet Katherine Scholz Burkett,
who cheerfully tolerates all this silliness...

## Important!

***Neither I, Doug Burkett, nor any of the contributors to this tip list are responsible in any way for any damage of any kind that you or anyone else may incur from using the information in this tip list. These tips are provided for educational purposes only. The contributors and I believe that all information is accurate and reliable, but we make no expressed or implied warrantee or guarantee to that effect. Some tips may involve hardware or software modifications to the calculator that may void the manufacturer's warrantee.***

This document is a collection of useful tips and hints relating to the TI-89 and TI-92 Plus calculators. The tips come from user experience and many other sources, and apply to all facets of calculator operation and programming. The tip list has grown to include reference material, humor and oddities, so perhaps the name *Tip List*, is no longer appropriate, but it stands for now.

The tip list is not a buglist, a wishlist or a FAQ. I maintain a wishlist, and good FAQs are available. While a FAQ is, by definition, frequently asked questions, this list instead describes useful techniques which may be 'infrequently asked'. These are sophisticated calculators and some features are not obvious. All users have different skills, and what is obvious to some is not obvious to all.

I want the tip list to be the ultimate, exhaustive source of TI-89 / TI-92+ operation. If you find a good tip or clever solution, please email me. In general I give credit to the first person to submit a particular tip. Let me know if your tip is from a manual, or you really learned it from someone else. If a tip  gives no credit, that only means I came up with it on my own, not that I am the original inventor.

Bhuvanesh Bhatt has generously volunteered to keep this list on his site:

http://triton.towson.edu/~bbhatt1/ti/

Thanks Bhuvanesh! Bhuvanesh also maintains a bug list for these calculators.

Through the gracious efforts of Andrew Cacovean, the tip list is also available here:

http://www.angelfire.com/realm/ti_tiplist/

Andrew developed and maintains this site. You can download the entire tip list or individual tips, tip list code and also the wishlist. Andrew has also written some very good examples of operation.

Some tips include code listings, which can be typed in. Most of the code is included in a zip file called *tlcode.zip*. I use lots of comments and white space in these listings, so you will save considerable RAM or archive memory by deleting these comments. Beginning with version 9.0, the code listings shown in the tips are not the same as those in the *tlcode.zip* file. The code shown in the tips contains explanatory comments. Those comments are not actually included in the *tlcode.zip* versions, to reduce RAM/flash requirements. Code from earlier tip list versions may still have all the comments included.

The tips are organized in these sections:

[1] *Miscellaneous*: Tips that do not fit in any other category, as well as humor and hardware
[2] *Computer Algebra System (CAS)*: Using the CAS for symbolic algebra
[3] *Data structures*: Working with matrices, lists and data variables
[4] *Graphing and plotting*: Function graphing, data plotting, operations on the graph screen
[5] *Graph-Link Cable and Software*: Using the TI GraphLink cables and software
[6] *Math*: Mostly numerical mathematics
[7] *TI Basic programming*: Programming techniques, tricks and work-arounds
[8] *String variables*: Manipulating string variables
[9] *User interface*: User input and output, dialog boxes and results formatting.
[10] *Units Conversions*: Using the built-in units conversions
[11] *Solving*: Using the various solving functions, including *solve()*, *nsolve()* and *csolve()*
[12] *C Programming*: A few tips on writing and using C and ASM programs

There are also a few appendices of reference material:

[A] *Anti-tips - things that can't be done*: Summary of calculator and CAS limitations
[B] *More resources*: Includes FAQs, TI documentation and useful web sites
[C] *Glossary*: of common and obscure terms
[D] *Quick reference guide*: to calculator commands and functions
[E] *Tip list programs and functions*: Alphabetic list of all routines in *tlcode.zip*

Note that the table of contents includes active links to all the tips.

I hope you find this useful or interesting, and that you will consider contributing your ideas and corrections to this list.

Doug Burkett
Eaton, OH USA
dburkett@infinet.com

# Contents

### 3.0 Data structure tips

### 4.0 Graphing and plotting tips

### 5.0 GraphLink Cable and Software Tips

## 6.0 Math Tips

## 7.0 TI Basic Programming Tips

## 8.0 String Variable Tips

## 9.0 User Interface Tips

## 10.0 Units Conversion Tips

## 11.0 Solving Tips

# Contributors

Contributors are credited in each tip. I would also like to list them here, and thank them again.

Andrew
Andy
Kenneth Arnold
Alex Astashyn
Bez
Bhuvanesh Bhatt
Billy
Damien Cassou
Andrew Cacovean
cj
Jordan Clifford
cybernesto
Martin Daveluy
Stuart Dawson
George Dorner
ES
Fabrizio
Larry Fasnacht
Glenn E. Fisher
Lars Frederiksen
Gp
John Gilbertson
Mike Grass
Frank Mori Hess
Titmité Hassan
Rick Homard
Sam Jordan
Paul King
Eric Kobrin
Kevin Kofler
Kosh DV
Ray Kremer
Cass Lewart
Daniel Lloyd
Christopher Messick
Olivier Miclo
Francesco Orabona
Roberto Perez-Franco
Mike Roberts
Rick A.
Samuel Stearley
TipDS
TM
Rafael Velazquez
Gary Wardall
Frank Westlake
Hank Wu
ZLX

## Revision record

*Various changes:*

Page numbering now section-local. New section 12, *C Programming Tips*. 'More resources material' divided into lettered appendices. New Appendix E, *List of programs and functions*.

*Contributors:*

Correct spelling of Samuel Stearley. New contributors: cybernesto, Carlos Becker, Francesco Orabona, Hernan Rivera, Rafael Velazquez..

*Tips changed:*

Introduction: Update Bhuvanesh' URL.
[1.16], [7.8], [7.36], [7.40] Update for new version of *copyto_h()*
[1.1] Links fixed
[1.10] Add more patents
[1.7] Add Bhuvanesh' method
[2.10] Considerably clarified, thanks to Carlos Becker
[2.13] Limit problem fixed in AMS 2.05
[3.4] Add *listrev()* function
[4.1] Added *when()* method
[6.4] Rename to *Gamma, log-gamma and factorial functions*. Add factorial function.
[9.5] Add method of saving defaults as a list
[9.14] Substantially changed
[B-1] Add actual TI FAQs contents, not just topics.
[B-4] New bibliography
[C] Many corrections and additions.
[D] Correct entries for *avgRC()*, *setMode()*, *Table, Logistic*.

*New tips added:*

| | |
|---|---|
| [1.15] | Permission to copy TI guidebooks |
| [1.16] | Use your TI-89 / TI-92 Plus as an on-screen ruler and protractor |
| [1.17] | Quick tips |
| [1.18] | Myth and legend from the TI discussion Groups |
| [1.19] | Kraftwerk's "Pocket Calculator" song |
| [1.20] | TI-92 Plus internal photographs |
| [1.21] | TI-89 internal photographs |
| [1.22] | TI Calc-Con '02 trip report |
| [1.23] | Ray Kremer interview on TI Student Feature page |
| [2.23] | Try *comDenom()* for faster partial factoring |
| [2.24] | Infinity^0 evaluates to 1 |
| [2.25] | Factor on complex i to show polynomial complex coefficients |
| [2.26] | *limit()* fails with piecewise functions |
| [3.24] | Use equations to build lists of lists and matrices |
| [3.25] | Internal error with *seq()* and Boolean operators in list index calculation |
| [3.26] | Find indices of specific elements in lists and matrices |
| [3.27] | Fast symbolic determinants |
| [3.28] | Fill a list or matrix with a constant |

| | |
|---|---|
| [3.29] | Convert data variables to matrices |
| [3.30] | Singular value decomposition (SVD) of a matrix |
| [3.31] | Store new list element at index greater than dimension |
| [4.14] | Completely clear Graph screen |
| [4.15] | Delete statistics plots from programs |
| [5.8] | Opening variables in GraphLink changes some characters |
| [5.9] | GraphLink software sends either TI-89 or TI-92 Plus files |
| [5.10] | Run GraphLink under Windows XP |
| [5.11] | Improve font size of printed program source code |
| [6.48] | Use *R>Pθ()* for four-quadrant arc tangent function |
| [6.49] | *Taylor()* polynomial function finds tangent line |
| [6.50] | The Savage Benchmark |
| [6.51] | Review of TI-89 / TI-92 Plus Calculus Tools |
| [6.52] | Return complex result in Real mode with 0i |
| [6.53] | Implicit derivatives |
| [6.54] | Delete variables after using numeric solver |
| [6.55] | Algorithms for *factor()* and *isPrime()* |
| [6.56] | Fourth-order splice joints two functions |
| [6.57] | Sum binary "1" digits in an integer |
| [6.58] | Extract floating-point mantissa and exponent |
| [6.59] | Accelerate series convergence |
| [6.60] | Faster, more accurate exponential integrals |
| [6.61] | Find more accurate polynomial roots |
| [6.62] | Try asymptotic expansion for functions of large arguments |
| [6.63] | Faster numerical matrix inverses |
| [6.64] | Calculate through undef and get correct results |
| [7.38] | Why use TI Basic? |
| [7.39] | Quickly delete locked, archived variables |
| [7.40] | Recall expression without variable value substitution |
| [7.41] | Indirection bug with local variables |
| [7.42] | Find variable names used in expressions |
| [7.43] | Faster function calls with list and matrix arguments |
| [7.44] | Local subroutines execute faster than global subroutines |
| [7.45] | Display text in multiple fonts, and Pretty-Print expressions |
| [7.46] | 'Custom' command makes menu labels for trapped function keys |
| [7.47] | Global variables not evaluated when local variable has the same name |
| [7.48] | Create functions in programs |
| [7.49] | Read and write text variables |
| [7.50] | Modify loop control variables within the loop |
| [8.6] | Convert strings to upper- and lower-case |
| [8.7] | Replacement for *mid()* and *right()* functions use position instead of count |
| [8.8] | Relational operators compare strings |
| [9.17] | Speed up the keyboard |
| [9.18] | Use multiple commands on entry line with ":" |
| [9.19] | Use a more efficient RPN interface |
| [9.20] | Use a graphical equation editor (equation writer) |
| [9.21] | Programs for simple equation libraries |
| [9.22] | Minimize TI-89 alpha entry inefficiencies |
| [11.12] | *nSolve()* may ignore solution constraints |
| [12.1] | Advantages of TICGG over TI SDK |
| [12.2] | Use C functions in expressions with AMS 2.05 |
| [12.3] | TI Basic extension app template |

*Changes to anti-tips:*

New anti-tips 17, 18.
Anti-tip 11: Fix reference to [9.11] to [9.10]

# 1.0 Miscellaneous tips

## [1.1] Discussion Group etiquette

TI maintains several discussion groups on their web site. These are like Usenet newsgroups in that you can ask questions or post comments, but unlike newsgroups, they are moderated by TI.

The discussion groups are organized in two categories, Calculators at

*http://www-s.ti.com/cgi-bin/discuss/sdbmessage.cgi?databasetoopen=calculators*

and Education and Curriculum at

*http://www-s.ti.com/cgi-bin/discuss/sdbmessage.cgi?databasetoopen=educationandcurriculum*

Ray Kremer developed these suggestions for etiquette on the discussion groups. If you follow the suggestions, you are more likely to get good, fast answers to your questions.

Groups:

- Try to choose the applicable group for your question. Graph Link problems in the Graph Link group, no TI-89 questions in the Comments and Suggestions group, etc.
- It is not necessary to post a question in multiple groups, the people likely to answer it frequent most of the groups and only need one post to respond to.

Before you ask:

- Check the manual for an answer to your question.
- Check tifaq.calc.org for an answer to your question.
- Check the FAQs at *http://education.ti.com/product/tech/89/faqs/faqs.html* and *http://education.ti.com/product/tech/92p/faqs/faqs.html* for an answer to your question.
- Look over the old subject headings for a question similar to yours, an answer to your question may already be in the group.

Subject heading:

- Put something in the Subject field, it's hard to get to the post otherwise.
- Use something short but descriptive of your question/comment. Do not just put "TI-89", or "HELP!!!".
- Do not say "please read", the people likely to have an answer read all the posts anyway.
- If you absolutely must make a very long subject heading, put a quotation mark " after the fourth word or so or else the whole thing will carry over in replies.

Author heading:

- Put something in the Author field, even a made up name. This makes it easier to tell who's talking if a back and forth exchange results.

Post body:

- If it is not a calculator specific group, specify the calculator you are using.
- Capital letters is considered shouting, turn that caps lock off.
- Do not say "please reply", if somebody can answer your question rest assured they will reply.

- If nobody answers your question it is usually safe to say nobody knows or there is no way to do what you ask.
- If you request that a reply also be e-mailed to you, do not say outright that you do not intend to return to the group.
- If you typed your address into the E-mail field, the person replying can see your address and send the response there by checking a box. So if you neglected to give your address the the body of your original post there's no need to post again just to give your address.
- Be as specific as possible. Give examples.
- Hit Submit only once. If you don't think it went through, open a view of the group in a new window and hit Reload just to make sure it isn't there.
- If English is your first language, even a half-hearted attempt at proper grammar, spelling, and punctuation is appreciated.

Threads:

- Do not start a new thread if your message is in response to another message. That's what the "Reply" button is for.
- If you hit "Submit Message" but don't get the "The following entry has been made:" screen, do not resend the message right away. Open the group page in a new window, hit the reload button on your browser, and check for the post you tried to make. Only if it's not there should you hit "Submit Message" again.

In addition to Ray's good suggestions, consider these:

Replies:

- Remember that people all over the world use the TI discussion groups. English is not the native language of many users, so please be tolerant of grammar, punctuation and spelling mistakes.
- Please be considerate in your replies. Flaming doesn't help anyone. Remember that you are responding to a real person. You might ask yourself "would I really say this, in person?"
- The newsgroups are limited to 300 posts, and old posts are not cached. Please try to avoid the temptation to respond to a post with just an insult, as this pushes older, useful posts off the group.
- Posts using profanity, vulgarity and obscenity will be deleted by the TI moderators.

*(Credit to Ray Kremer)*

### [1.2] TI-92 Plus manual error for Logistic regression equation

On page 150 of the TI92+ manual, the logistic equation is shown correctly as

```
y = a/(1 + b*e^(c*x)) + d            [right!]
```

However, on p236, the equation is given incorrectly as

```
y = c/(1+a*e^(-bx))                  [wrong!]
```

This has been corrected in the new combined manual for the TI-89 and TI-92 Plus.

### [1.3] Alphabetic list of reserved variable names

The TI-89 / TI-92 Plus have several system variables and reserved names. These can be found in the manual. However, the TI-92 Plus module manual list was not updated. The most up-to-date list of reserved names is found in the combined TI-89 / TI-92 Plus manual. That list shows the reserved names grouped by application type, and the list is not alphabetized. This makes it tedious to determine if a variable name is reserved. The table below shows all the reserved names, alphabetized. Variable names that start with greek letters are first in the table.

**TI-89 / TI-92 Plus Reserved Names**

Δtbl
Δx
Δy
Σx
Σx²
Σxy
Σy
Σy²
σx
σy
θc
θmax
θmin
θstep
c1 - c99
corr
diftol
dtime
eqn
errornum
Estep
exp
eyeθ
eyeϕ
eyeψ
fldpic
fldres
main
maxX
maxY
medStat
medx1
medx2
medx3
medy1
medy2
medy3
minX
minY
nc
ncontour
ncurves
nmax
nmin
nStat
ok
plotStep
plotStrt
q1
q3
R²
r1(θ) - r99(θ)
rc
regCoef
regEq(x)
seed1
seed2
Sx
Sy
sysData
sysMath
tØ
tblInput
tblstart
tc
tmax
tmin
tplot
tstep
u1(n) - u99(n)
ui1 - ui99
x̄
xc
xfact
xgrid
xmax
xmin
xres
xscl
xt1(t) - xt1(t)
ȳ
y1'(t) - y99'(t)
y1(x) - y99(x)
yc
yfact
ygrid
yi1 - yi99
ymax
ymin
yscl
yt1(t) - yt99(t)
zθmax
zθmin
zθstep
z1(x,y) - z99(x,y)
zc
zeyeθ
zeyeϕ
zeyeψ
zfact
zmax
zmin
znmax
znmin
zplstep
zplstrt
zscl
ztØde
ztmax
ztmaxde
ztmin
ztplotde
ztstep
ztstepde
zxgrid
zxmax
zxmin
zxres
zxscl
zygrid
zymax
zymin
zyscl
zzmax
zzmin
zzscl

## [1.4] Importing and exporting PC data with the TI-89 / TI-92 Plus

It can be useful to move data between a PC and the calculator. For example, you might want to make better data plots with a PC graphing program, from calculated results. Or you might want to download a large data set to the calculator for processing.

These suggestions apply to Windows PCs. Some tips for Mac follow.

Unfortunately, transferring data from a PC to the calculator is quite limited. There are only two types of data you can transfer from the PC to the calculator: text variables, and numeric matrices.

This tip comes from this TI document: *http://www.ti.com/calc/docs/faq/graphlinkfaq011.htm*

### *Sending text variables to the calculator*

1. Create the text in some application.
2. Use the standard Windows commands to copy the data to the clipboard. Even if the program Copy command does not explicitly mention the clipboard, the application most likely places a copy in the clipboard.
3. In GraphLink, choose File, New, and select the Data File type.
4. Use Edit, Paste to paste your text into the new text variable window.
5. Change the variable name from 'untitled' to your desired name.
6. Choose File, Save As, and save the text variable as a .9xt file.
7. Finally, select Link, Send, choose the text variable, select Add, then OK. The text variable will be sent to the calculator.

About all you can do with this text variable is view it in the text editor.

### *Sending matrix variables to the calculator*

Note that this tip does not work with GraphLink 2.1. TI has unfortunately removed the Tools, Import, ASCII Data menu item. It is no longer possible to send a matrix variable to the calculator. Hopefully TI will restore this feature in the future.

The basic principle is to convert the matrix data to a text file, then use GraphLink to import the file. The steps are:

1. Create a text file (.txt) that contains your data. The individual elements can be separated by spaces, commas, semicolons or tabs. From a spreadsheet, you can select the data range, then choose File, Save As, and select the text file type. The actual procedure depends on which spreadsheet you are using. It the PC application does not support saving data as a text file, you can copy the data to the clipboard, then paste it into a text editor such as NotePad.
2. In GraphLink, select Tools, Import, ASCII data. Choose the .txt file and select OK.
3. GraphLink shows a Save As dialog box. Save the file as a .9xm file, with a name that you choose.
4. GraphLink opens a new data window showing the matrix. If it looks correct, you can send the matrix to the calculator with Link, Send.

The GraphLink Import function is smart enough to recognize that the E character means exponential notation. If the text file data rows don't have the same number of elements, the matrix is created by padding the short rows with '0' elements.

If you are sending variables to use the linear regression functions or other statistics functions, use *NewData* to convert the matrix to a data variable.

### *Creating a matrix variable program with a spreadsheet*

If you have a PC spreadsheet program such ast Excel or Lotus 123, it can be used to create an ASCII text program which will create the matrix on the calculator. The data to be sent is arranged in the spreadsheet columns, and another column contains text strings and spreadsheet formulas which create the calculator matrix elements from the data in the spreadsheet cells.

The basic steps are:

1. Import the matrix data in the spreadsheet.
2. Enter the spreadsheet formulas which create an ASCII program in a spreadsheet column.
3. Cut and paste the ASCII program to a text editor, and save it as a text file.
4. Use GraphLink to import the ASCII program and send it to the calculator.
5. Run the program to create the matrix.

I will first describe the method for an Excel spreadsheet, then show how it is done with Lotus 123.

First,  put the matrix data in the spreadsheet starting at column A, row 7. If your matrix has more than one column, put the remaining matrix columns in spreadsheet columns B, C and so on.

Enter the ASCII program in an empty column. I will use column E as an example. Spreadsheet column E will contain the matrix creating program, in text form. Start by entering these text strings:

| | | |
|---|---|---|
| cell E1: | \start92\ | |
| cell E2: | \comment= | |
| cell E3: | \name=*prgm_name* | *(replace prgm_name with the program name)* |
| cell E4: | \file=*prgm_name*.TXT | |
| cell E5: | () | |
| cell E6: | Prgm | |

Cell E7 is a spreadsheet formula that creates a TI-Basic instruction, which in turn creates the first matrix row. If the matrix has three columns, contents of cell E7 are

```
+="["&TEXT(A7,"###.######")&","&TEXT(B7,"###.######")&","&TEXT(C7,"###.######")&"]"
&"/->/matrix_name"
```

The formula is entered as one line, not two as shown. Replace *matrix_name* with the name of the matrix variable you want to create on the calculator. The three TEXT functions convert the contents of cells A7, B7 and C7 to text strings. The "###.######" strings convert the cell contents to numeric strings, with six fixed decimal places. The string "/->/" is the ASCII TI-Basic equivalent for the *store* operator. For example, if *matrix_name* is *mat1*, and the contents of cells A7, Bu and C7 are 10, 20 and 30, then this formula creates the string

```
[10.000000,20.000000,30.000000]/->/mat1
```

Cells E8 and subsequent cells (one for each matrix row) define a formula which augments each additional matrix row. For a three-column matrix, cell E8 is

```
+="augment(matrix_name;"
&"["&TEXT(A8,"###.######")&","&TEXT(B8,"###.######")&","&TEXT(C8,"###.######")
&"]"&")/->/matrix_name"
```

If A8, B8 and C8 are 40, 50 and 60, then this formula creates the string

```
augment(mat1;[40.000000,50.000000,60.000000])/->/mat1
```

which augments the row [40,50,60] to *mat1*.

In the two cells in column E following the last *augment()* expression, enter these two strings to create the program footer:

```
EndPrgm
\stop92\
```

This table shows the spreadsheet to create a 3x3 matrix. The TI-Basic program name will be *makemat(),* and the matrix will be called *mat1*.

| | A | B | C | D | E |
|---|---|---|---|---|---|
| **1** | | | | | \start92\ |
| **2** | | | | | \comment= |
| **3** | | | | | \NAME=makemat |
| **4** | | | | | \file=makemat.TXT |
| **5** | | | | | () |
| **6** | | | | | Prgm |
| **7** | 10 | 20 | 30 | | [10.000000,20.000000,30.000000]\->\mat1 |
| **8** | 40 | 50 | 60 | | augment(mat1;[40.000000,50.000000,60.000000])\->\mat1 |
| **9** | 70 | 80 | 90 | | augment(mat1;[70.000000,80.000000,90.000000])\->\mat1 |
| **10** | | | | | EndPrgm |
| **11** | | | | | \stop92\ |
| **12** | | | | | |

Now copy the program in column E (from the \start92\ to \stop92\) and paste it in a text editor, such as Windows NotePad. Save the text file as makemat.txt. In GraphLink, choose Tools, Import, ASCII Program, and open makemat.txt. Send the file to the calculator and run it. The program *makemat()* can be deleted after the matrix is created.

To create the matrix program in Lotus 123, different formulas are required in cells E7 and E8. The formula for cell E7 is

```
+"["&@STRING(A7,6)&","&@STRING(B7,6)&","&@STRING(C7,6)&"]"&"\->\xyz"
```

and the formulas for cells E8 and below are

```
+"augment(xyz;"&"["&@STRING(A8,6)&","&@STRING(B8,6)&","&@STRING(C8,6)&"])"&
"\->\xyz"
```

The @STRING() function replaces the Excel TEXT function, and the format code of '6' specifies six fixed decimal digits. Otherwise, the functions are the same.

While this method works, there are some limitations. Different formulas are required depending on the number of matrix columns. You cannot use exponential notation formatting in the spreadsheet formulas, because the calculator will interpret the 'E' as a variable, instead of the exponent prefix. This is a serious limitation, but you can use the text editor 'Replace' command to fix the exponents.

You can also use this method to transfer list data to the calculator.

*(Credit to Stuart Dawson)*

### *Exporting data to PC applications from the calculator*

This is a simple matter of opening the variable in GraphLink, choosing Edit, Select All, then choosing Edit, Copy. This copies the variable contents to the clipboard. From there, they can be pasted into the PC application.

Or, in GraphLink, use Tools, Export, ASCII data to save the variable contents as a text file.

### *Importing and exporting with a Macintosh*

George Dorner offers these tips, if you are using an Apple Macintosh.

Moving pictures or data to a Mac:

This works as one would expect with drag and drop and copy/paste technology, first using Graphlink to create the variables of the correct type.

1. Create the text, matrix data, or graphic in the appropriate Mac application. I used SimpleText, a scanned Dilbert comic,  and Graphic Converter to experiment.
2. Copy the selected data (command C).
3. Open a new variable of the appropriate type with Graphlink at File->New (or command N). (Use .9xm,.9xl,.9xg,.9xs for matrices, lists, pictures, or strings.)
4. Paste the data to the screen which opens.
5. Save As <yourname>.9xt for a text file for example.
6. Drag and drop the new file to Graphlink. If you are in Auto mode, numbers will be taken as floating point. Change to Exact mode first if you want integers.
7. Use the variable as needed.

To size and crop a picture I use the shareware Graphic Converter from Thorsten Lemke.

Moving data from the calculator to your Mac application.

1. Open the variable in Graphlink.
2. Select  and copy.
3. Paste to the appropriate Mac application.

*(Credit to George Dorner)*

## [1.5] Creating picture variables on a PC

Picture variables have many practical uses on the TI-89 / TI-92 Plus.  For example, they can be used in programs to document the input variables pictorially. You can also use them as reference material, for example, for schematics, or for pinouts for integrated circuits.

While it is possible to create the picture variables directly on the TI-89 / TI-92 Plus, it is faster and easier to create them on the PC, then convert them to the calculator format. To convert them, use John Hanna's Image File Viewer program, which you can get here:

> *http://users.bergen.org/~tejohhan/iview.html.*

This description assumes that you are using a PC with Windows. If you have some other operating system, the basic procedure is the same. I welcome your comments on the procedure for other operating systems.

Pictures are displayed in the Graph window.

The general process is:

1. Create your picture using a PC graphics program, such as Paint which comes with Windows.
2. Convert the picture using Iview.
3. Download the variable to the calculator with GraphLink.
4. Crop the picture on the calculator if needed.
5. Upload the modified picture to the PC, using GraphLink, so you have a backup copy.
6. Archive the picture variable, if desired, to free some RAM.

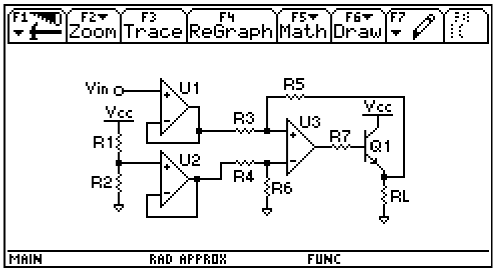
There are a few things you can do to make the picture creation process more efficient:

1. In the PC paint program, set the color depth to "black and white" or "monochrome". The calculator cannot normally display grayscale without assembler programs, so you won't be using colors or shades of gray.
2. Create a 'template' file on the PC, to use for creating a new picture. The template file contains all the symbols that you use in your drawings, and a rectangle of the size of the calculator display. For example, I have created a template file that contains electronic schematic symbols, and a rectangle that is 240 pixels wide and 91 pixels high. I use the rectangle to make sure that my picture will fit in the 92+ window.
3. Draw your picture in the template file. When you are satisfied with the appearance, cut it and paste it into a new file. This new file is the one that will be imported into the Image File Viewer program.
4. To put text in my pictures, I use the font called Small Fonts, with a point size of 5. This seems a nice compromise between readability and size, and results in text that looks good on the 92+ LCD display. Small Fonts has a filename smalle.fon, and is a Microsoft font, so it should already be on your PC.
5. Use the Image File Viewer 'Import' menu command to import the bitmap picture. Use the Save As menu command to save the picture as a .9xi file. Be sure to change the Pic Var Name to your desired variable name before using Save As, or the picture variable name will be the default "picture", regardless of the file name you entered.

After you have saved the picture as a .9xi file, use the Link, Send menu commands in GraphLink to send the file to the calculator. To display the picture manually, open the graph window with [DIAMOND]GRAPH. Open the picture with [F1] Open ..., then choose Picture as the type and select the variable name. Note that opening a picture doesn't clear the current contents of the Graph Window; you can use [F6] ClrDraw to do that.

Hanna's image file viewer program creates Picture variables that are the full size of the graph window. If you pictures are smaller than that, you can save lots of memory by cropping the picture to it's actual size. Here's how: open the picture variable, and select [F7], Save Picture. You are prompted for the 1st corner, which is the upper left corner. Use the arrow keys to move the first corner and press [ENTER]. You are now prompted for the 2nd Corner, so use the arrow keys to move the lower right corner, and press [ENTER]. Save the copy as a Picture variable with a new name. Delete the original picture variable.

This screen shot below show a schematic that I drew. Before cropping, this picture takes 3097 bytes. After cropping the picture to just the size needed, the picture takes about 1666 bytes, for a savings of 1431 bytes, or about 46%.



To use your pictures in your programs, read about these commands in the manual: *RclPic*, *AndPic*, *XorPic* and *RplcPic*.

### [1.6] Manual error for SetMode() codes for Angle modes

The on-line PDF verson of the *TI-89 / TI-92 Plus Guidebook* has an error in Appendix D, page 585. The codes for Angle should be

| | |
|---|---|
| Angle | 3 |
| Radian | 1 |
| Degree | 2 |

The manual mistakenly gives the Angle code as 0, and omits the codes for Radian and Degree.

### [1.7] Undocumented error codes

These error codes are not found in the *TI89/92+ Guidebook*:

| | |
|---|---|
| 305 | Expired product code |
| 435 | Invalid guess |

The text of these messages has changed for AMS 2.04/2.05:

| | |
|---|---|
| 165 | Batteries too low for sending or receiving<br>(Can now occur during variable transfers as well as Flash application transfers) |
| 490 | Invalid in Try..EndTry |
| 970 | Variable or Flash application in use |
| 980 | Variable or Flash application is locked, protected, or archived |

*(Credit to John Gilbertson)*

### [1.8] Make a protective plastic 'skin' for your calculator

If you use your calculator in a contaminating environment, or just while you're eating pizza, this tip offers some measure of protection.

Wrap the calculator with plastic wrap. Heat it with a hair dryer (blow dryer) on the medium setting, just until the plastic softens and conforms to the keys. Don't over-do it - the LCD display can be permanently damaged by too much heat.

*(Credit to Tip DS)*

### [1.9] Physical characteristics of TI-89 and TI-92 Plus

The table below shows some physical characteristics of the TI89 and TI92+. The characteristics for the TI-89 are for a HW1 calculator. Numbers in parenthesis refer to notes following the table.

| **Characteristic** | **TI-89** | **TI-92+** |
|---|---|---|
| Overall length (1)<br>Overall width (1)<br>Overall height (1) | 186mm (7.32 in)<br>89mm (3.50 in)<br>25mm (0.98 in) | 219mm (8.62 in)<br>125mm (4.92 in)<br>40mm (1.57 in) |
| Volume (2) | 414 cm3 (25.1 in3) | 1095 cm3 (66.6 in3) |
| Mass (with batteries & cover) | 264 g (9.3 oz) | 563 g (19.86 oz) |
| Mean density | 0.64 g/cm3 | 0.51 g/cm3 |
| Number of physical keys (3) | 45 | 80 |
| Key dimensions:<br>Number keys<br>Other keys<br>Function keys | <br>11 x 7 mm (0.43 x 0.28 in)<br>11 x 6 mm (0.43 x 0.24 in)<br>11 x 3.5 mm (0.43 x 0.14 in) | <br>9 x 5.5 mm (0.35 x 0.22 in)<br>up to 11 x 15 mm (0.43 x 0.59 in)<br>9 x 6 mm (0.35 x 0.24 in) |
| Key colors | White on gray<br>White on blue<br>White on black<br>White on yellow<br>Green on black<br>White on purple | White on gray<br>White on blue<br>White on black<br>Yellow on black<br>Green on black |
| LCD display physical width (4)<br>LCD display physical height (4)<br>LCD display area (4)<br>LCD display aspect ratio (4) | 55.5 mm (2.19 in)<br>34.5 mm (1.36 in)<br>19.1 cm2 (2.98 in2)<br>1.61 | 83 mm (3.27 in)<br>44.5 mm (1.75 in)<br>36.9 cm2 (5.72 in2)<br>1.87 |
| LCD display width in pixels<br>LCD display height in pixels<br>LCD display pixel pitch<br>Pixel count | 160<br>100<br>0.35 mm (0.014 in)<br>16,000 | 240<br>128<br>0.35 mm (0.014 in)<br>30,720 |
| LCD horizontal viewing angle (5)<br>LCD vertical viewing angle (5) | ±50 degrees from vertical<br>-15 to +50 degrees from vertical | ±50 degrees from vertical<br>±50 degrees from vertical |
| Batteries | (4) AAA<br>(1) CR1616 | (4) AA<br>(1) CR2032 |
| Enclosure fasteners | (6) Torx T6 fasteners | None (plastic slide lock) |
| Cover mechanism | Slide + snap | Snap,<br>cover can be used as tilt stand |
| External connectors | GraphLink I/O 2.5 mm | Graphlink I/O 2.5 mm<br>Overhead LCD |
| Agency approvals | FCC Part 15<br>Canadian Class B<br>CE<br>C-TIC (Australian) | FCC Part 15<br>Canadian Class B<br>CE<br>C-TIC (Australian) |
| Patent markings | 3819921<br>3921142<br>3932846<br>4115705<br>4208720<br>4348733 | (same as TI-89) |
| Country of manufacture | Taiwan R.O.C. | Taiwan R.O.C. |
| Copyright date | 1997 | None |
| Nameplate recess on back | Yes | No |

Notes:

(1) Measured at maximum dimension including slide-on cover
(2) Volume of maximum dimensions, not actual displacement volume
(3) 92+ cursor keypad counted as four keys
(4) Measured with respect to active pixel locations, not the full exposed display glass
(5) Very approximate, subjective measurements; depends on contrast setting, lighting and battery voltage

## [1.10] Patents for TI-89 and TI-92 Plus

This section contains abstracts of the six patents listed on the calculators, and additonal patents which may apply to graphing calculators in general. All the patents are assigned to Texas Instruments. For the full patent text including claims and figures, search by patent number at the US Patent and Trademark Office, here:

http://www.uspto.gov

Note that all the patents listed on the calculators issued between 1974 and 1982. I only present the abstracts, but it is the patent claims that specify the patented properties of the invention. Note also that two listed patents have the same title and abstract. This is not uncommon, and the claims section of the patents would describe the difference between the two inventions.

Jack Kilby is listed as an inventor on patent 3819921. Jack Kilby invented the integrated circuit, and was awarded the 2000 Nobel prize in physics for that work. For more information on Mr. Kilby, go here:

*http://www.ti.com/corp/docs/kilbyctr/jackstclair.shtml*

TI has been granted at least 89 patents with the word *calculator* in the title. If you are interested in the history of calculators in general and TI in particular, these may provide hours of interesting reading.

### 3819921: Miniature electronic calculator

Inventor(s): Kilby; Jack S., Merryman; Jerry D., Van Tassel; James H.
Issued/Filed Dates: June 25, 1974 / Dec. 21, 1972

Binary-coded decimal electronic calculator capable of adding, subtracting, multiplying and dividing with some degree of automatic decimal point placement to provide a visual display of answers of up to 12 decimal digits. The decimal digits are serially displayed at a speed compatible with the calculator operations. The parts of the calculator are so adapted electrically and mechanically in relation to each other to result in a minature portable battery operated calculator of extremely small dimensions for example the outside case dimensions of 41/4 inches by 61/8 inches by 13/4 inches and very low weight of about 45 ounces, having a calculating capability only before obtainable in calculators of much larger size and weight while retaining mechanical and operational simplicity. Some significant aspects of the calculator are the primary electronics embodied in an integrated semiconductor circuit array located in substantially one plane for performing the arithmetic calculations and generating the control signals, a keyboard input arrangement located in substantially one plane parallel to the integrated semiconductor circuit array for producing unique electrical signals corresponding to number and command entries and a visual display using a semiconductor array, as for a thermal printer for printout.

### 3921142: Electronic calculator chip having test input and output

Inventor(s): Bryant; John D., Hartsell; Glenn A.,
Issued/Filed Dates: Nov. 18, 1975 / Sept. 24, 1973

An MOS/LSI semiconductor chip for providing the functions of an electronic calculator includes a data memory, an arithmetic unit for executing operations on data from the memory, and a control arrangement for defining the functioning of the machine including a ROM for storing a large number of instruction words, an instruction register for receiving instruction words from the ROM and reading out parts to various sections of the control arrangement, and an address register for selecting the location in the ROM for read out of the next instruction. Input and output terminals are provided for keyboard input, display output, timing signals, etc. A test mode of operation is provided for quality control upon completion of manufacture of the chip. The test mode allows the entire ROM to be tested by reading in addresses to the address register from external and reading out the resulting word from the instruction register. During the test mode, normal incrementing and branching of the address register may be externally inhibited.

**3932846: Electronic calculator having internal means for turning off display**

Inventor(s): Brixey; Charles W., Hartsell; Glenn A., Vandierendonck; Jerry L.
Issued/Filed Dates: Jan. 13, 1976 / Sept. 24, 1973

An electronic calculator system of the type having a keyboard input and a visual display is implemented in MOS/LSI semiconductor chips having a data memory, an arithmetic unit, a read-only-memory for storing instruction words, and control circuitry for operating the system in response to keyboard inputs by selecting addresses for instructions from the read-only-memory, all of which is located in monolithic semiconductor units. A technique is provided for turning off the display after a selected time period by holding an instruction word in an instruction register while repeatedly incrementing an address register for the ROM until it overflows, then branching to an address defined in such instruction word. This is repeated until the selected time period is reached.

**4115705: Electronic calculator with push-button on-off system**

Inventor(s): McElroy; David J.
Issued/Filed Dates: Sept. 19, 1978 / June 14, 1976

An electronic calculator with a power supply ON-OFF arrangement actuated by momentary-closure push-button switches which are part of the keyboard. A bistable latch circuit on the calculator chip is continuously powered by the battery, and is caused to flip to an ON condition by actuating an ON key, and this turns on a large, low-resistance transistor which is in series with the voltage supply line going to all of the other electronic circuitry on the chip.

**4208720: Calculator with algebraic operating system**

Inventor(s): Harrison; Herman W.
Issued/Filed Dates: June 17, 1980 / July 26, 1976

Disclosed is an electronic calculator having a data entry unit for inputting numeric data, expressions such as parentheses and hierarchal mathematical commands, an arithmetic unit for performing arithmetic operations on the numeric data, a memory for storing the numeric data and associated hierarchal mathematical commands inputted via the data entry unit and logic circuitry for enabling the arithmetic unit to perform arithmetic operations on numeric data inputted via the data entry unit within a pair of parentheses, the logic circuitry enabling the arithmetic unit to perform a higher order hierarchal mathematical command before a lower order hierarchal command even though the higher order command is received after the lower order hierarchal mathematical command.

**4348733: Calculator with algebraic operating system**

Inventor(s): Harrison; Herman W.
Issued/Filed Dates: Sept. 7, 1982 / Nov. 19, 1979

Disclosed is an electronic calculator having a data entry unit for inputting numeric data, expressions such as parentheses and hierarchal mathematical commands, an arithmetic unit for performing arithmetic operations on the numeric data, a memory for storing the numeric data and associated hierarchal mathematical commands inputted via the data entry unit and logic circuitry for enabling the arithmetic unit to perform arithmetic operations on numeric data inputted via the data entry unit within a pair of parentheses, the logic circuitry enabling the arithmetic unit to perform a higher order hierarchal mathematical command before a lower order hierarchal command even though the higher order command is received after the lower order hierarchal mathematical command.

The following patents are not listed on the calculator, which means that they may not include claims which apply to the TI-89 and TI-92 Plus. However, it is good practice to mark the product with *all* relevant patents, because this aids in proving willful infringement during litigation, with its attendant treble damages.

**4,521,868 Electronic data processing apparatus with equation operating system having improved exponentiation operators**
*[possibly related to 89/92+/V200's entry line and numeric solver]*

Inventor(s): Caldwell; David; Ferrio; Linda J.; Hunter; Arthur C.
Issued/Filed Dates: June 4, 1985 / February 29, 1984

The equation operating system is an improved data and command entry format together with a compatible operation system for use with electronic data processing apparatuses, most particularly with scientific calculators which provide an alphanumeric display of entered equations. This invention provides a different set of commands and displayed characters for performing exponentiation. An up arrow together with a down arrow serve to define the expression of the exponent. The numeric data stored in a numeric display register upon implementation of the exponentiation is raised to the power of the expression between the up arrow and the down arrow. This invention also includes an integral power exponentiation command which enables easy entry of single digit positive integral exponents, thereby eliminating the need for a completing command.

**4,823,311 Calculator keyboard with user definable function keys and with programmably alterable interactive labels for certain function keys**
*[TI-95 keyboard, but also applies to an extent to 89/92+/V200]*

Inventor(s): Hunter; Arthur C.; Ferrio; Linda J.
Issued/Filed Dates: 18 April 1989 / May 30, 1986

Calculator having a keyboard in which one or more keys have labels created by a display and subject to changing interactively as the user desires. Typically, advanced scientific-programmable calculators may have too many functions to be adequately included on the keys of the keyboard associated therewith. In such calculators, certain functions require a plurality of keys to be actuated in order to be performed. Thus, such keyboards tend to be cluttered an confusing to the user. Thus, a keyboard is proposed having a small number of keys labeled with different functional labels as the user proceeds through a menu or tree structure containing all the desired functions. Keys in a certain group of keys on the keyboard are thereby subject to redefinition or relabeling so as to provide a variety of functions.

**5,168,294 Display demonstration system for providing enlarged projected image of calculator display**
*[magnifying device for View Screen]*

Inventor(s): Davis; Peter H.; Christensen; Brad V.; Ahlfinger; Robert R.
Issued/Filed Dates: September 27, 1991 / December 1, 1992

A calculator or other computing device uses a remote display set upon an elevated platform above the base of an overhead projector. Light through the base lens is passed through the display and is enlarged by lens.

**5,392,081 Calculator projection display**
*[ViewScreen]*

Inventor(s): Tarnay; Thomas N.; Wyatt; W. Gerald
Issued/Filed Dates: February 21, 1995 / September 29, 1993

A projection display for an electronic data processing device for use in conjunction with an overhead projector is provided. The projection display includes a display with a light transmitting screen, and the screen is framed by a uniquely constructed housing. The housing defines a free convection air channel below the screen to conduct a cooling air flow. The air flow in the free convection air channel substantially eliminates the problem of image deterioration due to screen overheating.

**5,532,946 Calculator with table generation capability**
*[TI-82 table, feature also on 89/92+/V200]*

Inventor(s): Phipps; Ann E.; Santucci; David M.

Issued/Filed Dates:  July 2, 1996 / September 11, 1995

A digital computer or calculator is equipped with a numerical data table generation capability. It provides a user with the ability to specify one or more mathematical functions, and the ability to specify how the numerical data in support of the functions are to be displayed.

**5,377,130 Method and apparatus for solving terms of a numerical sequence**
*[sequence graphing mode solving algorithm]*

Inventor(s): Frank; Olivier L.; Phipps; Ann E.

Issued/Filed Dates: January 26, 1993 / December 27, 1994

A digital computer is provided for generating a solution of a term of a numerical sequence. The digital computer includes a memory. The memory has stored within it solving instructions of a sequence-solving program and recognizing instructions for recognizing the type of numerical sequence. The digital computer also includes a processor for executing the solving instructions and recognizing instructions. An input device receives data from a user. The data represents a mathematical expression of the numerical sequence. The data also represents a value of a term identifier representing the term of the numerical sequence to be solved. A display is also provided for displaying the solution of the term.

**5,870,319 Device and method for collecting data from graphed images**
*[developed for, but not necessarily limited in application to, TI graphing calculators]*

Inventor(s): Thornton; Glen Allen; Ferrio; Linda Jean; Stone; David S.; Howard; Veronica L.
Issued/Filed Dates: February 9, 1999 / January 4, 1996

A computing device for capturing designated data from one or more graphic applications comprising a screen area for viewing one or more graphic functions wherein each function is manipulated with a plurality of shortcut keys or a cursor pad communicably linked to the screen area. A first applications with a screen interface for the user is provided within the device having graphing capabilities for manipulating the graphs on the screen area. A cursor, cursor pad and shortcut keys are used to trace data along the graph and capture the points to a stack area where they are stored for future use by a second application. The second application gives the user the ability to further study and analyze the data. Alternatively, shortcut keys can be used to automate the transfer from the first application to the second application.

*(Credit to John Gilbertson for the recent patents)*

## [1.11] "Low Battery" indication voltages

The TI89/92+ gives a low-battery voltage indication in the display status line in the lower right corner of the display. There are two levels of low-battery indication. The first level at the highest battery voltage is displayed as "BATT". The second level at a lower voltage is displayed as "BATT" in inverse video (white characters on a black background). This table shows the battery voltages for a HW1 TI-89 and a HW2 TI-92+.

| | TI-89 HW1 | TI-92+ HW2 |
|---|---|---|
| First "BATT" indication | 4.13V<br>1.03 V/cell<br>69% of 6V | 4.35 V<br>1.09 V/cell<br>73% of 6V |
| Second "BATT" indication | 3.85V<br>0.96 V/cell<br>64% of 6V | 4.11V<br>1.03 V/cell<br>69% of 6V |

The percentages shown in the table indicate the percentage of the 'fresh battery' voltage, assuming that each of the four cells is initially at 1.5V. This information may be useful if you are considering powering your calculator with rechargeable NiCd batteries or some other type of rechargeable battery.

## [1.12] Processor clock speed and battery voltage

The TI89/92+ use an RC (resistor-capacitor) network to set the speed of the processor clock. This means that the processor clock frequency is not set as precisely as it would be if a crystal was used. Some people have claimed that this results in clock frequency variations as the batteries discharge during use.

I tested the variation in clock frequency as a function of battery supply voltage by timing a short test program at various voltages from 6V down to the second low "BATT" indication. I found much less than 1% variation in clock speed for both a HW1 TI-89 and a HW2 TI-92+.

GraphLink transfers, particularly AMS upgrades, are know to fail at low battery voltages. One theory has been that these failures are caused by clock speed variations that result in synchronization failure.

My test results do not support this theory. However, during my testing I did notice that, at very low battery voltages, the calculator will turn itself off while the program is running. This effect may cause the GraphLink transfer failures. Another possible cause is that low battery voltages reduce the voltage levels of the GraphLink signals such that they no longer meet the minimum requirements for successful transmission.

### [1.13] Developer's guide describes calculator internal operation

The SDK developer's guide, formally called the *TI-89 / TI-92+ Developer Guide*, is a 1400-page document which describes much of the internal operation of the calculator. While this document is intended to enable software developers to write calculator applications, it is interesting reading even if you have no intention of writing applications, but you are just curious as to how your calculator works. While registration is required to download the complete SDK, you may download just the guide, without registration, at

http://education.ti.com/developer/8992/down/download.html

### [1.14] Cleaning a calculator after spills and other accidents

Accidents happen even if you are careful with your calculator. If you spill a drink or liquid into your calculator, such as milk, carbonated beverage or juice, it should be cleaned out, otherwise damage will probably result. Short-term failures include complete failure, sticky or inoperative keys, or intermittent display operation. Even if no immediate failure is evident, the spilled liquid can cause long-term failures through corrosion.

If your calculator is under warrantee or you are not confident in your ability to disassemble your calculator, consider sending it to TI for repair or replacement. Most users have reported that TI is fairly cooperative in working with them.

There is some risk in the cleaning procedure, but if your calculator has failed completely, you have nothing to lose, anyway.

The key principles for a successful cleaning job are to clean the calculator as soon as possible after the spill, clean it thoroughly, and dry it completely and quickly. The purpose of the cleaning process is to remove the contaminants from the calculator components with damaging the calculator. All batteries *must* be removed before attempting a cleaning.

Clean water is probably the best fluid to use to clean a calculator. Other solvents such as alcohol may damage the calculator. While it may seem strange to wash electronics with water, this is common in the electronics industry. At my firm, we have been washing printed circuit boards after assembly, in a dishwasher, for over fifteen years. We have fairly stringent reliability requirements, and the washing process improves reliability by removing the solder flux before conformal coating.

This process has been successful in cleaning a variety of spills:

1. If the calculator seems to be working, back up the calculator memory with GraphLink.
2. Disassemble the calculator. See instructions below.
3. Gently scrub the printed circuit board (PCB) with warm water and a clean toothbrush. Rinse frequently. Use distilled water, or softened water, because hard water will leave calcium deposits. Alternatively, you can secure the parts to be washed in a dishwasher, and run the

dishwasher for about ten minutes. Remove the key caps as needed, and wash and rinse them. Wash the circular gold patterns on the display side of the PCB, but try to keep the water away from the display. Wash the rubber keypad sheet; the black bumps make the switch contacts and need to be cleaned well. You may need to pull back the foil shield sheet to thoroughly clean the printed circuit boards (PCBs).

4. Gently dry the washed components with a hair dryer. Avoid over-heating the printed circuit board and plastic parts. The goal of this step is to quickly evaporate the water without damaging the parts from excessive heat.
5. Let the calculator dry for at least 24 hours.
6. Reassemble the calculator. See instructions below.

The instructions below apply to a HW1 TI-89 and a HW2 TI-92+. They may not apply to other hardware versions.

***Disassembly and assembly instructions for the TI-89***

You will need a small Phillips screwdriver (#0), a Torx T6 driver, and a small flat-blade screwdriver to pry out back-up battery.

Disassembly:

1. Place the calculator face down on a towel or other soft surface.
2. Remove the battery access cover and the four AAA cells.
3. Remove Phillips screw over back-up battery door, then remove the door. Remove the back-up battery. The battery is spring-loaded, so place your finger over the battery as you pry it out, or it may go zinging up in the air, like mine did.
4. Remove the six T-6 Torx fasteners around the outer edge of back of calculator.
5. Remove the back plastic cover: squeeze the edges in at the top of the calculator and pull back until this end pops out. Then gently wiggle it back and forth while pulling back, and the back will pop off fairly easily.
6. Remove the two phillips-head screws holding shield foil.
7. Hold the shield foil back and remove the four phillips-head head screws, under foil. These four screws hold the printed circuit board (PCB) in.
8. Gently remove the PCB. The key caps are loose, so keep the front of the calculator facing down, or they will fall out. Remove any key caps which are stuck to the rubber key sheet, and replace them in the calculator front piece.

Assembly:

1. Install the rubber keypad sheet. Note that the rubber sheet only installs one way. Press all the key 'bumps' into the key caps.
2. Install the PCB in the front cover. Ensure that the black plastic flap at the upper right of the PCB is tucked inside case. Make sure all key caps are still in place.
3. Install the two phillips-head screws through foil shield.
4. Install the four phillips-head screws that hold the PCB.
5. Install the plastic back piece. Align the front and back piece and press them together until they snap closed.

6. Install the six Torx screws.
7. Install the backup battery, (+) side up, and the backup battery door.
8. Install the AAA batteries and the cover.
9. Check the memory with Var-Link, and restore if needed. Note that even if the memory contents seem complete, custom units and custom key assignments may be lost.

***Disassembly and assembly instructions for the TI-92+***

You will need a T-8 Torx driver and a small Phillips screwdriver or flat-blade screw-driver. A T-9 driver can also be used.

Disassembly:

1. Remove the back plastic cover. Put the calculator face-down on a soft surface.
2. Remove the four AA batteries.
3. Remove the back-up battery clip and backup battery. If the backup battery is held in too tightly, bend the end of a heavy paper clip and use this to pry out the battery.
4. Loosen the 14 T-8 screws which hold the black inner cover. You do not need to completely remove the screws fromt the cover. Lift of back inner cover.
5. Remove the T-8 screw through the foil shield at the upper left of the back-up battery area.
6. Lift out the PCB.
7. Remove the rubber keypad sheet.

Assembly:

1. Install the rubber keypad sheet. Make sure that the plastic alignment pins are through the holes in the keypad sheet.
2. Install the PCB.
3. Install the short T-8 screw through the foil shield.
4. Install the inner back cover and align the screws with the mating holes, then tighten the 14 T-8 screws
5. Install the back-up battery, (+) side up, and the battery cover. Tighten the battery cover screw.
6. Install the (4) AA batteries.
7. Install the back outer cover and fasten the lock.
8. Check the memory with Var-Link, and restore if needed. Note that even if the memory contents seem complete, custom units and custom key assignments may be lost.

## [1.15] Permission to copy TI guidebooks

This information is from *http://education.ti.com/global/copy.html.*

**Permission to Copy TI Guidebooks and Teacher Guides**

Texas Instruments (TI) grants you permission to copy and distribute, but not modify, all or any portion of any user support documents on this web site (http://www.education.ti.com). You can use this material for the purpose of preparing classroom instructional materials and educational articles for publication.

Use or distribution for any other purpose is expressly prohibited without written permission from Texas Instruments.

*Permission to use TI materials for other purposes*

To obtain written permission from TI to use or distribute TI materials for purposes other than those described above, contact:

Texas Instruments Incorporated
7800 Banner Drive, MS/3918
Dallas, TX  75251
Attn: Manager, Business Services

*Requirements for using TI materials for other purposes*

Once you have obtained permission to use TI materials for purposes other than those described above, you must include acknowledgement of the TI copyright in any new materials that you create. The acknowledgement must be in the following form:

If the entire document is reproduced, the credit lines at the beginning of the reproduction should read:

*Reproduced with permission of the publisher.*
*Copyright 20xx Texas Instruments.*

Note: Insert the correct copyright year for xx.

If any part of a document is reproduced, credit lines should appear immediately after the extracted material and should read:

*Material extracted from (Document Title) with permission of the publisher.*
*Copyright 20xx Texas Instruments.*

Note: Insert the correct name for Document Title and the correct copyright year for xx.

If the material is extracted from several documents and consolidated in the same reproduction:

- Assign sequential reference numbers to the individual documents used.
- List the documents with the copyright information by the reference number at the end of the reproduction.
- Refer to the documents immediately after the extracted materials by number only.

For example:

At the end of your document:

1. Material extracted from (Document Title) with permission of the publisher.
   Copyright Texas Instruments.
2. Material extracted from (Document Title) with permission of the publisher.
   Copyright Texas Instruments.

In the text of your document:

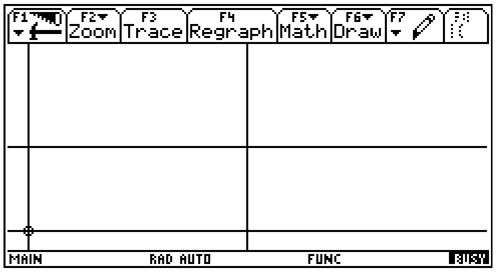
See Ref. 1, or
Extracted from Ref. 2.

TI makes no warranty regarding the accuracy of the information presented in the respective manuals, including fitness for use for your intended purpose. Also, TI is under no obligation to update the information presented in these manuals.

## [1.16] Use your TI-89/TI-92 Plus as an on-screen ruler and protractor

The program in this tip, called *ruler()*, lets you measure small objects on the calculator screen. You must be careful not to damage the screen, but with some care it works fairly well. The program also shows some useful programming techniques such as automatically identifying the calculator type and doing some simple graphics. TI Basic is fast enough for this simple interactive program to be usable.

This screen shot shows the measurement screen.



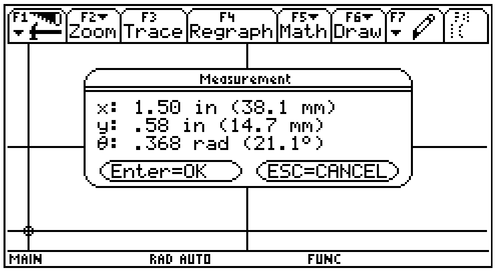
The object is measured between two sets of cross-hair cursors. The *origin* cursor at the lower left screen corner is identified by the small circle around the cross-hair intersection. The *target* cursor is at the center of the screen. Both cursors are moved with the blue cursor keys. When *ruler()* is first started, the cursor keys move the target cursor. The [O] key toggles origin and target movement.

These are the key functions, which can be displayed by pressing [H], for 'help'.

| | |
|---|---|
| [O] | Switch between target movement and origin movement. When the program is started, the target can be moved. Push [O] to move the origin, then push [O] again to move the target. Repeat as necessary. Push [-] on the TI-89, don't use [alpha]. |
| | |
| [UP] | Move cursor up one pixel |
| [2ND] [UP] | Move cursor up ten pixels |
| [DOWN] | Move cursor down one pixel |
| [2ND] [DOWN] | Move cursor down ten pixels |
| [LEFT] | Move cursor left one pixel |
| [2ND] [LEFT] | Move cursor left ten pixels |
| [RIGHT] | Move cursor right one pixel |
| [2ND] [RIGHT] | Move cursor right ten pixels |
| | |
| [M] | Display the measurement. Push [5] on the TI-89. |
| [STO] | Copy the measurement to the home screen |
| | |
| [H] | Display the help screen. Push [8] on the TI-89. |
| [HOME], [ESC], [QUIT] | Exit the program |

If the cursor moves beyond the edge of the screen, it will appear at the opposite screen edge. This is called wrap-around.

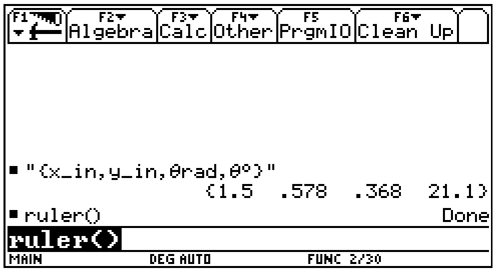
The measurement is shown by pressing [M], as this screen shot illustrates:



```
Measurement
x: 1.50 in (38.1 mm)
y: .58 in (14.7 mm)
θ: .368 rad (21.1°)
```

The *x* measurement is the horizontal distance between the two cursors. The *y* measurement is the vertical distance between the two cursors. The measurements are negative if the origin is above the target or to the right of it. The angle $\theta$ is measured from the origin cross-hair to the target. The angle is undefined (*undef*) if x = 0 and y = 0.

Push [STO] to copy the current measurement results to the home screen. This screen shot shows the result after exiting *ruler()*:



```
■ "{x_in,y_in,θrad,θ°}"
                    {1.5  .578  .368  21.1}
■ ruler()                              Done
ruler()
```

The string in the entry column shows that the list elements are the x-measurement, the y-measurement, and the angle measurement in radians and degrees. You can extract the individual elements with *list[n]*, where *n* is 1 to 4. For example, to extract the y-measurement, use the cursor keys to highlight the list, press [ENTER] to copy it to the entry line, then type [2] [ENTER], and .578 is entered in the history. This feature is accomplished with the *copyto_h()* utility, which must be installed in a folder named *util*.

One version of *ruler()* runs on both the TI-89 and the TI-92 Plus. This table shows the measurement limits for the two calculators.

| | **TI-89** | **TI-92 Plus** |
|---|---|---|
| Maximum x-measurement | 2.18 in (55.4 mm) | 3.29 in (83.6 mm) |
| Maximum y-measurement | 1.06 in (27 mm) | 1.42 in (36 mm) |
| Approximate x, y resolution | 0.0138 in (0.4 mm) | Same |

The accuracy is limited by my ability to accurately measure the pixel pitch, as well as screen parallax, LCD manufacturing consistency and the size of the pixel. Total error may be as much as two or three pixels, so the accuracy cannot be better than about 0.05 inches. The angle resolution depends on the distance between the target and the origin. If x = 5 pixels and y = 5 pixels, the resolution is about 5°. The best resolution is about 0.3° for either calculator.

*Source code description*

The source code for *ruler()* is shown on the following pages. I have added comments to describe the operation, but these comments are not in the program in the *tlcode.zip* file.

*ruler()* uses local programs and functions so it can be distributed as a single program. This does require using global variables to pass parameters, since the scope of local variables defined in *ruler()* does not extend to the other locally-defined programs. The global variables are stored in the *main* folder, since it is guaranteed to exist on every calculator, and they are deleted before exiting *ruler()*.

Since *ruler()* needs to change the Graph and Mode settings, they are saved in the beginning of the program and restored at the end. The Angle mode is set to Radian, so I know that the results of angle calculations are in radians. To change the Graph and Mode settings, I use the numeric string arguments for *setGraph()* and *setMode()*, instead of the text strings. This makes program operation independent of language localization.

I want the graph screen clear except for the cursors, so this code completely clears and displays the graph screen:

```
    ClrDraw                       © Clear all plots, functions and drawings
    ClrGraph
    PlotsOff
    FnOff
    SetGraph("3","1")             © Turn grid off
    SetGraph("4","1")             © Turn axes off
    DispG                         © Display graph screen
```

To avoid writing, distributing and maintaining two versions of the program, I detect which calculator model (TI-89 or TI-92 Plus) on which the program is running. The model is determined by using *getConf()* to find the calculator screen width. If the width is 240, then the calculator is a TI-92 Plus, otherwise it is a TI-89. The expression to find the model is

```
    if getconfg()[dim(getconfg())-14]=240 then
    ...
```

The test returns *true* if the calculator is a TI-92 Plus. *getConfg()* returns a list of configuration pairs. The location of the screen width configuration pair depends on whether or not certificates are installed. However, the screen width is 14 elements from the end of the list, regardless of certificate installation, so the expression gets the configuration with respect to the end of the list, that is, 14 elements from the end.

Three specific parameters depend on the calculator model: the screen size, the key codes for some keys, and the pixel pitch. I account for all these parameters by assigning the correct values to variables based on the calculator model, then the program logic uses these variables instead of hard-coded constants.

I use the letter keys [O], [M] and [H] for features in *ruler()*. The same keys are used for both the TI-89 and the TI-92 Plus. However, on the TI-89, I don't make the user enable alpha mode to press the right key. Instead, I test for the unmodified key code for that letter key. For example, the [O] key is the same as the [-] key, so I just test for the [-] key code (45), instead of the [O] key code (79). The TI-89 user presses the [-] key to perform the origin/target toggle function.

The user can press any of three keys to exit the program: [ESC], [QUIT] or [HOME]. This just makes it easier for the user, since she does not need to remember exactly which key stops the program. The cursor movement keys are consistent with the standard operation on the Graph screen. Pressing just the cursor key moves one pixel, and pressing [2ND] with the cursor key moves the cursor a larger step.

There are a few dedicated function keys, and they are not necessarily obvious, so I include a 'help' screen, which is shown each time the program starts.

*ruler()* has two shortcomings which could be remedied. First, it is not obvious whether the origin or the target cursor will move when the cursor keys are pressed. However, it is obvious which cursor moves once the key is pressed, and the other cursor can be selected by pressing [O]. Second, the program would be easier to use if the measurement was continuously shown in the status line. This cannot be done with TI Basic, but can be done with a C program. I chose not to do that, in order to keep the program simple.

*Source code for ruler()*

```
ruler()
Prgm
© On-screen ruler & protractor
© 15julØ2/dburkett@infinet.com
© Calls util\copyto_h()


© Global variables

© main\µox     Origin cursor x-coordinate
© main\µoy     Origin cursor y-coordinate
© main\µtx     Target cursor x-coordinate
© main\µty     Target cursor y-coordinate
© main\µx      Maximum x-coordinate
© main\µy      Maximum y-coordinate
© main\µscl    Scale factor, inches/pixel


© Local variables

local gdb,key,draw_all,mode,help,move_x,move_y,modedb,p,,qØ1,qØ2r,ko,km,kh,khm,dist

© gdb          graph database settings
© mode_db      Mode settings
© key          pressed-key code
© ko           [O] key code
© km           [M] key code
© kh           [H] key code
© khm          [HOME] key code
© mode         Cursor mode: 1 == move target cursor, Ø == move origin cursor
© p            Measurement results
© qØ1          Results label string for [STO] (copy results to home screen)
© qØ2          Results string for [STO] (copy results to home screen)
© r            Angle measurement string
© dist         Function: calculate measurements
© draw_all     Program: draw both cursors on Graph screen
© help         Program: display 'help' screen
© move_x       Program: move target or origin cursor in x-direction
© move_y       Program: move target or origin cursor in y-direction


©==========================================
© Local program and function definitions
©==========================================

© draw_all()
© Draw both cursors on LCD screen
```

```
Define draw_all()=Prgm
 PxlHorz main\μoy
 PxlVert main\μox
 PxlHorz main\μty
 PxlVert main\μtx
 PxlCrcl main\μoy,main\μox,2
EndPrgm

© help()
© Display help dialog box

Define help()=Prgm
 Dialog
  Title "Ruler Help"
  Text "[O] Set origin"
  Text "[M] Measure"
  Text "[STO▸] Copy measurement to Home"
  Text "[H] Help"
  Text "To quit:"
  Text "[HOME], [ESC] or [QUIT]"
 EndDlog
EndPrgm

© move_x(mode,distance)
© Move cursor (target or origin) in x-direction

Define move_x(m,i)=Prgm
 If m=1 then                         © Move target cursor
  PxlVert main\μtx,0                 © ... erase old cursor line
  mod(main\μtx+i,main\μx)→main\μtx   © ... find new coordinate with wrap-around
 Else                                © Move origin cursor
  PxlVert main\μox,0                 © ... erase old cursor line
  PxlCrcl main\μoy,main\μox,2,0      © ... erase old origin circle
  mod(main\μox+i,main\μx)→main\μox   © ... find new coordinate with wrap-around
 Endif
EndPrgm

© move_y(mode, distance)
© Move cursor (target or origin) in y-direction

Define move_y(m,i)=Prgm
 If m=1 then                         © Move target cursor
  PxlHorz main\μty,0                 © ... erase old cursor line
  mod(main\μty+i,main\μy)→main\μty   © ... find new coordinate with wrap-around
 Else                                © Move origin cursor
  PxlHorz main\μoy,0                 © ... erase old cursor line
  PxlCrcl main\μoy,main\μox,2,0      © ... erase old origin circle
  mod(main\μoy+i,main\μy)→main\μoy   © ... find new coordinate with wrap-around
 Endif
EndPrgm


© dist()
© Find distance and angle for current cursor locations

Define dist()=Func
 local dd,dr,xd,yd
 © dd  angle in degrees
 © dr  angle in radians
 © xd  x-axis distance between cursors
 © yd  y-axis distance between cursors

 (main\μtx-main\μox)*μscl→xd         © Find x-axis distance
 (main\μoy-main\μty)*μscl→yd         © Find y-axis distance

 If xd=0 and yd=0 Then               © Find angle
  undef→dr:dr→dd                     © ... angle is undef if x=0 and y=0
 else
  R▸Pθ(xd,yd)→dr                     © ... else calculate angle in radians
  dr*180/π→dd                        © ... and degrees
 EndIf
 return {xd,yd,dr,dd}
EndFunc


©=================
© Begin mainline
©=================

StoGDB gdb                    © Save graph database settings
GetMode("ALL")→modedb         © Save mode settings
setMode("3","1")              © Set Angle mode to radians

ClrDraw                       © Clear all plots, functions and drawings
ClrGraph
PlotsOff
FnOff
SetGraph("3","1")             © Turn grid off
SetGraph("4","1")             © Turn axes off
DispG                         © Display graph screen

© Initialize calculator-specific parameters

if getconfg()[dim(getconfg())-14]=24Ø then   © Determine calculator model by screen width
 239→main\μx                  © For TI-92+:  set maximum x-dimension
 1Ø3→main\μy                  ©              set maximum y-dimension
 111→ko                       ©              set [O] key code
 1Ø9→km                       ©              set [M] key code
 1Ø4→kh                       ©              set [H] key code
 8273→khm                     ©              set [HOME] key code
 .Ø13772→main\μscl            ©              set scale factor in inches/pixel
else
 159→main\μx                  © For TI-89:   set maximum x-dimension
 77→main\μy                   ©              set maximum y-dimension
 45→ko                        ©              set [O] key code
 53→km                        ©              set [M] key code
 56→kh                        ©              set [H] key code
 277→khm                      ©              set [HOME] key code
 .Ø1381→main\μscl             ©              set scale factor in inches/pixel
endif

© Initialize cursor positions

1Ø→main\μox                   © Set origin cursor 'x' 1Ø pixels from screen left
main\μy-1Ø→main\μoy           © Set origin cursor 'y' 1Ø pixels above screen bottom
intdiv(main\μx,2)→main\μtx    © Set target cursor to center of screen
intdiv(main\μy,2)→main\μty

1→mode                        © Set mode to move target cursor

draw_all()                    © Redraw cursors
help()                        © Display 'help'

© Main loop

Loop
 GetKey()→key                                © Get pressed key code
 If key=264 or key=khm or key=436Ø:Exit      © Exit program

© Handle cursor movement keys

 If key=34Ø Then              © [RIGHT]
  move_x(mode,1)              © Move cursor 1 pixel right
  draw_all()                  © Update screen

 ElseIf key=337 Then          © [LEFT]
  move_x(mode,¯1)             © Move cursor 1 pixel left
  draw_all()                  © Update screen

 ElseIf key=338 Then          © [UP]
```

```
  move_y(mode,¯1)             © Move cursor 1 pixel up
  draw_all()                  © Update screen

 ElseIf key=344 Then          © [DOWN]
  move_y(mode,1)              © Move cursor 1 pixel down
  draw_all()                  © Update screen

 ElseIf key=4436 Then         © [2ND] [RIGHT]
  move_x(mode,1Ø)             © Move cursor 1Ø pixels right
  draw_all()                  © Update screen

 ElseIf key=4433 Then         © [2ND] [LEFT]
  move_x(mode,¯1Ø)            © Move cursor 1Ø pixels left
  draw_all()                  © Update screen

 ElseIf key=4434 Then         © [2ND] [UP]
  move_y(mode,¯1Ø)            © Move cursor 1Ø pixels up
  draw_all()                  © Update screen

 ElseIf key=444Ø Then         © [2ND] [DOWN]
  move_y(mode,1Ø)             © Move cursor 1Ø pixels down
  draw_all()                  © Update screen

© Handle feature keys

 ElseIf key=ko Then           © [O] Toggle origin/target adjustment mode
  when(mode=Ø,1,Ø)→mode       © If mode = Ø, toggle to 1 and vice versa

 ElseIf key=kh Then           © [H] Display 'help' screen
  help()

 ElseIf key=km Then           © [M] Display measurement results
  dist()→p                    © Calculate measurements
  if p[3]=undef then          © Set angle measurement display string
   "undef"→r                  © ... handle 'undef'
  else                        © ... else format radian and degree results
   format(p[3],"F3")&" rad ("&format(p[4],"F1")&"°)"→r
  EndIf

  Dialog                      © Display measurements
   Title "Measurement"
s   text "x: "&format(p[1],"F2")&" in ("&format(25.4*p[1],"F1")&" mm)"
   text "y: "&format(p[2],"F2")&" in ("&format(25.4*p[2],"F1")&" mm)"
   text "θ: "&r
  EndDlog

 elseif key=258 then          © [STO] Copy measurements to Home screen
  "{x_in,y_in,θrad,θ°}"→qØ1   © Save label ...
  dist()→qØ2                  © ... and measurements
  util\copyto_h("qØ1","qØ2")  © ... then copy them to the Home screen
  draw_all()                  © ... and redraw to clean up after copyto_h()
 EndIf

EndLoop

©==================================
© Clean up before exiting program
©==================================

                              © Delete global variables
delvar main\µox,main\µoy,main\µtx,main\µty,main\µx,main\µy,main\µscl
rclGDB gdb                    © Restore graph database settings
setMode(modedb)               © Restore mode settings
DispHome                      © Display Home screen

EndPrgm
```

### [1.17] Quick tips

This section summarizes things that perplex some new users. If you read this one page, you'll be able to use your calculator much more effectively.

1. To run a program, type in the program name followed by "()", then press [ENTER]. For example: game() [ENTER]. If the program is not in the current folder, type the folder name with a backslash before the name, for example: main\game() [ENTER].
2. Make sure the GraphLink cable is plugged in firmly to the calculator connector. Push hard. The TI-89 has this problem more than the TI-92 Plus.
3. Use new batteries when upgrading the AMS with the GraphLink, or the upgrade may fail, and then you will have to install the AMS from a computer.
4. The latest version of the GraphLink software from TI may fix linking problems. Get it here: http://education.ti.com/product/accessory/link/down/download.html. Also make sure you have chosen the correct cable type (black or gray) and the right serial port.
5. If you want exact fraction results (like 1/3), press [MODE] and set Exact/Approx to Exact or AUTO. For approximate fractions (like 0.3333), set Exact/Approx to APPROXIMATE. Or, include a decimal point with one of the numbers: 1./3. Or, press [DIAMOND] [ENTER] instead of [ENTER].
6. There are no cube root or nth-root keys or functions. To find the cube root of *x*, use x^(1/3). To find the nth root, use x^(1/n).
7. There is no key for 'log', the base-10 logarithm. To find log(x), type in [L] [O] [G] [ ( ] [x] [ ) ] [ENTER]. Or select *log(* from the [MATH] menu.
8. To find the base-b logarithm of *x*, use $\log_b(x)$ = log(x)/log(b) = ln(x)/ln(b).
9. To plot a vertical line in the Y= function editor at x=a, use y1(x)=when(x<a,-9E999,9E999)
10. There are no built-in functions for secant, cosecant and cotangent. You can define these functions yourself.
11. There is no built-in interpolation function. Define your own interpolation function with *Define intrp(xa,xb,ya,yb,x)=ya+(x-xa)/(xb-xa)*(yb-ya)*. *xa* and *xb* are the x-values, *ya* and *yb* are the y-values, and *x* is the value at which to interpolate for *y*.
12. Temperature conversions are not in the UNITS menu. Instead, use `tmpCnv()` (to convert temperatures) and `ΔtmpCnv()` (to convert temperature ranges or differences).
13. The STAT VARS screen (ShowStat command) shows sample standard deviation, not population deviations. Display the population standard deviation by entering σx or σy. To enter the σ character, use [DIAMOND] [ ( ] [alpha] [S] on the TI-89, or [2ND] [G] [S] on the TI-92 Plus.
14. To find double and triple integrals, nest integral functions, for example, `∫(∫(f(x,y),y),x)`.
15. On the TI-89, the 'space' character is [alpha] [ (-) ].
16. There is no built-in quadratic equation solver. Use zeros(a*x^2+b*x+c,x)
17. Previous results stored in variables can cause wrong results. Press [F6] (for the Clean Up menu), then use Clear a-z or NewProb to delete old variables.
18. The small manual that comes with the calculator does not cover all features. For the full manual, go to http://education.ti.com/global/guides.html. You can also buy a printed manual from TI.
19. ticalc.org has games.
20. Most TI-89 programs will run on the TI-92 Plus and vice versa. Exceptions are ASM programs, programs which use specific key codes, and programs which use the larger TI-92+ display.

## [1.18]  Myth and legend from the TI Discussion Groups

*"The duty of Old Timers: Since there is*
*no known archive of these discussion*
*groups,  there ought to be a written history."*
*- oark*

*"Alright, you asked for it. Just keep in mind*
*that early written histories spent generations*
*as oral histories before someone wrote them*
*down, and thus they became exaggerated,*
*even mythical in nature."*
*- Ray Kremer*

These posts are mostly from the TI Off Topic discussion group. The intent is humorous, although just how much may depend on your familiarity with obscure events and running jokes. I decided to archive them here, because posts are not kept for long on the discussion groups and TI does not make archives available to us.

Ray Kremer keeps a collection of these legends at http://tifaq.calc.org/legends/.

### *The TI-liad*

by Ray "Homer he ain't" Kremer [2:06, 12/19/01]

Many moons ago, in the land of Suggestions, somebody started a thread entitled "How's the weather?" and great multitudes responded in answer to the question. And the great creator Paul King looked down from Mount Texas and he found it humorous, and he created a new land and called it "How's the Weather?" And Paul King said to the peoples of the world, "This new land is an experiment, and it may last for a long time or for only a short time."

The sage Ray Kremer saw the new land, and he said to the people, "There has been much Off Topicing in the land of 89 and 92+, and it has crowded out the 89ing and the 92+ing.  But look!  Here is a new and fertile land in which we can Off Topic to our hearts' desire!"  The people agreed with the sage, and many came to the new land and there was much joyous Off Topicing.

The evil demoness Laquisha Bonita from the underground realm of the Trolls saw the people frolicking in the new land, and grew jealous of their happiness, and she came to the new land and rebuked Paul King for creating it with her blashphemous cries of "Oh no you don't!".  And the peoples fought against the demoness, and told her "Nay, this is a good land, and we praise the great Paul King for creating it."  But Laquisha used her satanic ghetto-speak and met all arguments with threats of bodily harm, and spent much time telling how she would stomp on all our heads.

Soon many of the people tired of fighting against Laquisha, but the valiant knight Captain Ginyu vowed to continue the battle until the end.  And great flames arose from the war between Laquisha and Ginyu, and they spread across the land and spoiled it.  Then Paul King once again looked down from Mount Texas, and he saw that the new land he created was spoiled by the great war, and he saw that jokes from before the war had been carried by trickster spirits and brought into the land of 89 and 92+ where they become in-jokes, and he became sorrowful.

So Paul King called to his people and told them, "I have given you this fertile land, and see how you lay waste to it. The experiment is over!"  And Paul King cast down his lightning bolt caused the new land to sink into the sea, and the peoples scattered from it and returned to whence they had come.  The demoness was appeased by this, and she vanished, and the world was once again at peace.  But some say that she lurks still in the shadows, ready to appear when her name is invoked.

## *The TI-liad, continued*

by oark [3:24, 12/19/01]

An oracle told me this:

"The Creator will lift his protection of obscurity causing the gates of the Dark Realm to open once again. An army of trolls will be unleashed upon the new land discovered by Kremer. King Paul will then conduct the Second Great Purging; it will lead to the land's destruction.

"Kremer will reboard his ships and visit many strange groups, surviving perils and suffering much before he finally arrives in the land of the 89 and the 92 Plus. Many will say to him: 'GIVE ME GAMES. HERE IS MY EMAIL. AND TELL ME HOW TO MAKE GAMES IN PLAIN ENGLISH THIS TIME!!!!! DONT GIVE ME A WEB PAGE. I ALREADY READ THAT AND IT DIDNT MAKE SENSE. I NEED INSTRUCTIONS IN ENGLISH!!!'

"His patience will be legendary, but not his courage. When he encounters pirates from the land of HP, he will cower and refuse to proclaim the superiority of the TI-89. The gods will look down from Mount Texas with anger and bring Kremer to his knees with a 2KB ASM limit. Many from his own land will suffer and blame him for their frustrated math programming efforts.

"Following Kremer's unceremonious return, he will remain in service as a wise man whom many will consult. Nevertheless, commoners will treat him with a disgusted gratitude. Kremer's Limit will give power back to the defeated HP Empire; it will rise from the ashes of the HP 49G and conquer the educational market. HP zealots such as SS will come to power and burn the books detailing Kremer's famous works. And so Kremer will be lost to a history untold."

This doesn't sound very good for you Ray. I advise you to cheat fate and defend the TI-89 and TI-92 Plus lest these things befall you. However, you do seem to be a central figure after the Troll War. however.

## *The TI-liad, continued (even more)*

by Doug Burkett [12:53, 12/19/01]

<<And Paul King cast down his lightning bolt caused the new land to sink into the sea, and the peoples scattered from it and returned to whence they had come.  The demoness was appeased by this, and she vanished, and the world was once again at peace.>>

Part deux ...

"Yea, the loyal and the faithful were scattered and without form, wailing and shrieking in the wilderness. And in the low, grey lands, Ray of Kremer was broken. But he rose, and he toiled in the dark black arts of aitchtee-em-ell, and he learned of the spells and incantations of the black lords of the Server, whom all must serve. And one day he spoke the runes and cast the bones, and the lightening lit the night skies, and the thunder rolled without end, and Ray of Kremer cried "Behold! From the void, I have created the new land of OffTopic for you, my brothers and sisters. Rejoice and be glad, but bloody well behave yourselves this time!"

And though this new bright land be not on the maps of middle earth, and be not listed in the rolls of the Server, it does be, none the less. And King Paul, darkest and most powerful lord of the Server, said "Oh, all RIGHT!" (or something like that). And the faithful and penitent journeyed and pilgrammaged to this promised land, though it be distant and inaccessible. And as with one voice, they cried "Hail, Ray of Kremer, for he has made our new home!" And Ray of Kremer gazed upon them, and said, one more time, "Read this:" And they were glad and rejoiced.

And so it came to pass that one day, a vision appeared on the plains, and the vision was the goddess High-D, with visage terrible and beautiful, and the brothers and sisters were afraid, and fell prostrate and quaked. But she said to them "Rise, and be not afraid! For though you are innocent and work and play and post in your land of OffTopic, the dark lords of the Server are ever watchful with their baleful One Eye, and their vengeance is swift and without mercy. So play nice! :)"

So ends the reading.

And here's a bit about another historical event some of you may remember:

"And it would often come to pass that the mild and abiding ways of the people were often disturbed and cast asunder by the trolls, spammers and twits, especially in the deep nights of the days of rest. And their posts would be foul, and profane, and certainly off-topic, and they reveled and cavorted in an orgy of wanton abandon. And their obscene destruction did deeply offend the good people, even such that some of good will did weaken and take part, though they regretted it much.

"And in the deepest, darkest hour of night, the DG was all waste, and there was naught but nonsense. And one of the common  people, with no name, watched, and became angry, and cold, and said 'this is my home; I will not surrender it to these base cowards'. And he loosed the flood, and it rolled over the DG, and one by one the posts of the debasers were dispatched to the void, and though the debasers were legion and fought back with evil, he persevered wordlessly against their attacks, and he was victorious.

"And the debasers returned not for those days of rest, and the good people ventured back, and they asked 'who among us has made this flood, and for good or ill?', but he was silent."

So ends the reading.

## *Legend of the great war*

by Doug Burkett [15:09, 12/21/01]

In the dark times before IC, the common people did toil under the burden of calculating diverse sums, and products, and roots and special functions. And some did have the magick rules that slid, and they did toil less, and some fewer did have the noisome Friden, or Marchant, or Monroe, and some fewer still could petition the keepers of the big iron, who would mostly laugh and spit at them, unless they could show great wealth or a student ID.

And so the common people were without champion, until emperor Kilby, of the TI empire, had mercy and decreed to his slaving minions, that there would be a pocket calculator. And the minions were confused, and they said "a tiny mathematician?", for in those days a calculator was one who calculated, and not a machine. And emperor Kilby was short with them, and said unto them, 'no, you twits! a machine which calculates!' And the minions did say 'oh, OK, gotcha', and they did go forth and toil, and toiled some more, until there came forth the beast with the eight baleful red eyes, and it was named the DataMath 2500, but the ways of the marketing witches are strange and without reason. And lo! the common people could add, and subtract, and multiply, and divide, either from batteries or the AC line, and there was much rejoicing.

But those among the common people who did toil at mathematics, and science, and engineering, and the other arcane arts; they did sniff and say 'verily, but my rule which slides doth more than this, and surely without electricity to boot'. And so they were not amused. But then one among them, one emperor Hewlett of the HP empire, had mercy and decreed to his slaving minions, that there would be a scientific pocket calculator. And those minions did toil, and toil even more, and the beast HP-35 did in good time issue forth. And it was named for the number of keys it had, because, in those quaint times, the marketing witches did not hold sway in the HP empire. And all the subjects of the empire saw that it was good, very good in fact, and they danced and made merry in their caves, and cried 'Rejoice, and be glad! for now we shall surely crush the TI infidels, much like small insects!'

But it came to pass that the TI subjects were not, in fact, crushed like small insects. For instead their leaders held council, and consulted with sages and marketing witches, who told them 'Go forth and market your tiny mathematicians to those apprentices who study, for while they may have student IDs, they have not great wealth, and they lust after the little boxes with red numbers, that they may spend less time in study and more time making merry. And heed not the anquished wails and howling of those who teach, their cries will come to naught as does chaff on the wind.'

And thus did holy jihad commence, and the empires of HP and TI did do battle most bloody and vile. Both empires sent multitudes of beasts and riders into the fray, and verily it came to pass that the HP empire did prevail with those who toiled for coin at the arcane arts, and the TI empire did prevail with the young apprentices. This, even though the beasts of TI were weak of display and key, and often died early in battle. But they were legion, and they were cheap, and the apprentices had not great wealth, etc, etc.

And jihad raged on, and a great many beasts passed on to their just rewards, with their shields or on them, until the fateful day the HP empire unleashed a leviathan, and the name of the leviathan was 48SX, and 48SX had not a one-line eye, but an eye of hundreds of pixels, and 48SX did not just tedious calculations with numbers, but manipulated the magick symbols of Al-Jabr, and 48SX did not just speak of number, but painted and drew and sang, and communed much with larger calculating beasts. And those of the battle plain were stricken dumb, for all knew the leviathan was the true king. So once again the leaders of the TI empire consulted much with their marketing witches, who did divine and labour frantically at the entrails of small animals, and cast many spells. And the witches did say 'Lo! though the teachers of apprentices wail and lament with vigour at your reign, you may yet bewitch them with their own spells. For though they fear the calculating beasts, they do lust after tenure and job security. Do go forth and beguile them with with many free calculating beasts, and seduce them with much, much swag, and speak often at their covens in their tongue. Verily, you will befriend and comfort and succor them greatly, and they will sing your praise on high to the acquisition committees.' So this the TI emperors did, and at great length, with much expense in time and coin, and the prognostications of the marketing witches did come to pass, and the TI empire did flourish.

Still, the true HP king and his heirs did rule for many generations, and his subjects were legion and loyal, even though they could only speak to the king in his native tongue of arpean. But the years did pass, and the issue of the king's loins was not distinguished, and was but a pale imitation of the king, and the HP emperors were bewitched, and slumbered, while the subjects toiled and proclaimed mightily in the name of their king, and built mighty works in his honor.

But the empire of TI did not slumber, and the minions toiled, and toiled some more, and little by little built a leviathan after their own hearts. And the name of this leviathan was 92, with eye bigger than 48SX, and brain faster than 48SX, and memory deeper than 48SX, and knowledge broader than 48SX. And thus it came to pass that 92 was a worthy adversary to 48SX, and begat his progeny 89, who was much the same but could ride in a really big pocket instead of a small chipcart.

And thus began the golden age of the calculating beasts, who could and would even do battle with the lumbering behemoths of the Peasea empire. But still did jihad rage most violent with the subjects of the HP and TI empires, tearing brother from brother and sister from sister in sworn allegiance to the empires. But the HP emperors slumbered on, and on, and could not be roused by the clamor and cries of the HP subjects.

But in the deserts of Oz, the HP emperors did rouse themselves, one more time, just barely enough to summon noble wizards and clever sorcerers and true believers, who toiled mightily in the caves of Ayceeoh, and did mighty battle with the HP marketing witches, who now wielded all great power in the HP empire. And they did prevail, and the new HP king, 49G, was born, and at first his subjects rejoiced and danced in the newgroups, and verily, thumbed thier noses at the TI subjects in a most insulting manner. But the new king was much as the old king, in bright shiny dress, albeit with deeper memory and broader knowledge. And the HP subjects were thrown into turmoil and confusion, and some fought one against another as thier consciences ordered, and some defected to the TI empire, and some defected all empires to wander in alone the wilderness.

But the HP emperors would not have commerce with the teachers of apprentices, and were in fact aloof to them and snubbed them. And the HP wizards were slow to share all secrets of the new king, at length in massive tomes, and indeed said only 'read the tomes of the old king.' And thus the teachers did not hail the new king 49G one tiny bit, and many loyal HP subjects shunned the new king, and he languished in disgrace and ingominy, for verily his buttons required the strength of ten stout men to get his attention. And thus the new HP queen Carly of Fiorina and her marketing witches wailed and lamented, and to thier eternal shame and disgrace, they cast out forever the noble wizards and clever sorcerors who begat the new king.

And thus ended the golden age of the calculating beasts, not with a bang, but with a whimper.

But still the jihad rages on, though the emperors themselves now heed it not, and consider the jihad as the tiresome buzzing of insects in the thunderstorm. And the subjects of the empires see not that they are truly of one calling, greater than TI or HP, and thus they rail at each other and not at the true foe, and weaken themselves and bring on their own anguish and ruin.

So ends the legend.

## *The Battlefields of the Great War*

Translated by Ray Kremer

And it took place that during the Golden Age, one of the fronts of the great jihad was the land of 89 and 92+ in the Realm of the discussion groups. It was mostly peaceful there, but often a new soldier would approach the land and ask which army he should choose to fight in.  And this would stir the warriors of the jihad into battle once more, and the battles would last until the warriors were using the whip on their deceased horses.

The peoples were sorrowful at dwelling in the war ravaged land, and they cried out to the great creator Paul King to provide another battlefield in order to save the land of 89 and 92+ from harm. And Paul King from atop Mount Texas answered the prayers of the people and created the land of comparisons.

Thereafter most of the battles took place there, and the land of 89 and 92+ was saved from the horrors of war. And through the end of the Golden Age, and even a bit after it ended, the jihad carried on in the battlefield of comparisons, and the warriors fought each other valiantly on the fields were the bones of many fallen heros lay.

## *Legends of the discussion groups: "The dark wizard and the magik runes"*

by Ray "Homer he ain't" Kremer [20:16, 12/22/01]

One peaceful day at the end of the week-time, a dark wizard came to the Realm of the discussion groups. He had discovered a weekness in the Realm, that the fields of topik were vulnerable to spells of html, for the magik runes of angle brackets passed through the fields as they were without conversion into the codes of ampersand. The dark wizard took advantage of this knowledge and began to wreak havoc in the lands of the Realm, bringing forth unnatural talismans that turned the lands dark and fearsome.

In one land the dark wizard attempted a powerful spell of Javascript, but the magik was too powerful for the wizard to contain and the spell swept over the land as a great fog, and anyone who entered the land was unable to see anything. Undaunted, the wizard traveled to another land and cast spells that caused to appear great images of women of loose morals. The wizard thought this a great prank, but the peoples of the land were filled with dismay. The great sage Ray Kremer saw the evil that the wizard had done there, and to thwart the wizard he copied the spell of Javascript, and cast it with errors as the wizard had, and a great fog fell upon that land too, and it hid the images along with everything else.

Soon came the beginning of the next week-time, and from atop Mount Texas the great creator Paul King looked down and saw the evil that had been wrought, and he cast down his lightning bolt upon the thread that were infected with the spells of html, and he placed a great spell of protection upon the fields of topik so that they never again would be affected by the magik runes of angle brackets. The dark wizard, with his powers removed from him, was never heard from again.

But the fields of topik are not protected from all magik runes, for the marks of quotation are still taken wrongly by the fields when a re-ply is made upon a post.

## *Legends of the Discussion Groups: "The Re-creation"*

by Ray "Homer he ain't" Kremer [0:30, 12/23/01]

It came to pass in the ancient year of mmi that the creators looked down from Mount Texas and said to themselves, "Our domain has become stagnant. We should beat it down with cataclysms and rebuild it anew." And there was much agreement, and they set about at the task of re-creation.

Finally they finished, and the world was much changed from what it had been. And from the peoples in the Realm of the discussion groups came a great wailing and gnashing of teeth, and they cried out "The great cities of the world have been rearranged, and we know not the lay of the land, and it is impossible to find anything."

The appearance of the world was changed too. Gone were the colors of the sky and ocean. They were replaced with the colors of fire and the autumn. And from the peoples in the Realm of the discussion groups came a great wailing and gnashing of teeth, and they cried out "Our eyes cannot bear to look upon the world now. Surely we will all vomit if we view the altered lands any further."

But the creators atop Mount Texas were satisfied with their labors, and they closed their ears to the complaints of the people.

Not very long after this, the creators decided that the road they had built to lead apprentices to their temple was dull, and they feared that many apprentices would lose interest along the way. So they sent the malevolent and omnipotent goddess tispt, who is known to some by another name, to rebuild the road. So tispt and her divine servants tore down the old road, and built a new road filled with many flashing lights and shiny objects in order to hold the fleeting attention of the apprentices. And they also placed upon the new road a marketplace where free amusements could be found, and also maps to other such marketplaces. The apprentices rejoiced at this, for the creators has never before placed any emphasis on the amusements. New lands in the Realm of the discussion groups were also built, these for giving aid to the apprentices in the areas of their studies. And the peoples of the Realm found these works of tispt humorous, and they scoffed at them at first, but they had forgotten what it is like to be a young apprentice. And the toils of tispt were successful, and all worked as it was intended, and the road became traveled by many apprentices.

## *Legends of the Discussion Groups: "The Meeting"*

by Author Unknown [5:41, 12/23/01]

And it came to pass that the gentle and naive subject, Lalu of the CAS-Votaries Gild, requested the honour of the malevolent and omnipotent goddess tispt's presence at a chance encounter, high atop Mount Texas. And the goddess tispt did consult her wise counsel, goddess High-D (with visage terrible and mischievous), who did advise that such an encounter might compromise the ancient tee-ay code of secrecy of selfdom. The goddess tispt avowed the truth of the words her co-goddess had spake, but alas, she was not wholly swayed by the pronouncement of her wise, yet obstinate, counsel. Thus ensued a longsome deliberation twixt the two obdurate goddesses.

At long last, a resolution was realized, harsh and stern though it were, that no mortal would ever contemplate the countenance of the leaders of the empire of tee-ay in the temple in which they dwelt. Verily, the goddess tispt did decree to the gentle and naive Lalu, and to all of his guileless supporters, that no such encounter should ever take place, for such a meeting of mortal man and everlasting sovereign would surely result disastrous. "Go forth, and be merry in your ignorance of the ways of the leaders!" ordained the goddess. "You are but mere lackeys and shall never know the ways of the sovereign rulers!"

But the gentle Lalu proved not so naive, and his pursuit of the encounter perdured, that he might one day savour, if but for a moment, the sweet wisdom and phantasy and goodwill that floweth, as a deluge, from the summit of Mount Texas. And his imploring was like a psaltery upon the goddess's ears. And the resistance of the malevolent goddess did weaken at the insistence of Lalu. And anew did she seek the counsel of the goddess High-D, and again did the goddess High-D determine that mere mortals would ne'er able bear the ribaldry and sapience of the sovereign rulers. But the malevolent and omnipotent goddess tispt was not so easily convinced this tyme. And a debate ensued.

Yet the hour of Lalu's arrival drew nigh, and still the goddesses had not reached unfaltering agreement. Hour after hour did they wraxle with the idea, and yet they came to no resolution. Yea, even now it is rumoured that the deliberation persists, and the question of whether the two - mortal and goddess - ever did meet is left to the annals of history.

## *Legends of the Discussion Groups: "The Meeting", continued*

by unknown [6:06, 12/23/01]

Yes, they did meet ;)))

## *Legends of the discussion groups: "The Platter of Amusements"*

by Ray Kremer [2:56, 12/27/01]

In the olden days there were three great cities of amusements. There was the city of Tee Aye Fyles, but the years took their toll on that city and it fell into ruins, and eventually even the rubble where it had stood could not be found. There was also Dymension Tee Aye, later to be known as Calk of Org. (The great Library of Fak was located nearby, but a short ride from the city walls.) But by far the oldest, largest, and most famed of the cities was Tee Aye Calk of Org, and many apprentices did walk through the gates of that city.

It happened that the city of Tee Aye Calk of Org came to the attention of the lords of Mount Texas, and calling down from on high they said to the Governors of the city, "Deliver to us a selection of your best amusements, and we shall incribe them upon magik platters and disperse them among the apprentices of the land, and their love for us will increase as will the fame of your city." The Governors did as they had been asked, and the lords of Mount Texas did as they had promised, and as they had foreseen the apprentices did praise the lords and visit the city in greater numbers than ever before.

Soon though, dark forces whispered in the ears of the great giants of Pa-Rence, and the giants found upon the platters things that made them red-eyed with anger, and they attacked Mount Texas with great clubs and the looming threat of unleashing the most feared unholy creature ever to be spawned in the darkest depths of the underworld, the Lawyer. The lords of Mount Texas then did hurl their lightning bolts and destroyed every platter they could find, at great cost and difficulty to themselves.

And the Governors of Tee Aye Calk of Org did tremble at the terrible thing they had caused, and they felt much anguish knowing that the lords of Mount Texas had been terribly inconvenienced by their foolishness, and they evacuated the city and closed its gates while they searched for the things which made the giants angry and burned them on the pyre. And while the city gates were closed their was a great wailing and gnashing of teeth from the apprentices, who were too foolish to make do with the amusements they had already collected or to seek out other places where they could be found.

Soon though, the Governors of Tee Aye Calk of Org finished their search and reopened the city, and the apprentices once more filled its walls. And the lords of Mount Texas learned to be more careful when dealing with mortals.

## *Legends of the discussion groups: The Great Libraries*

by Ray "Homer he ain't" Kremer [2:43, 12/28/01]

The great sage Ray Kremer dwelt long in the Realm of the discussion groups, but it was not always so. In time long forgotten, Ray Kremer did aquire for the first time a ship that could surf the Sea of Webb, and during his voyages he came upon the Realm of the discussion groups and found there many apprentices in great need of guidance. And the great sage Ray Kremer, who had read the ancient tome of Manual, was able to preach sermons to the apprentices that set them on the right path. But new apprentices who had not heard the sermons were constantly arriving in the Realm, and Ray Kremer did often repeat his sermons for their benefit. One of the most popular was the sermon of the cord of Graff Lynk, but it is a very long sermon, and Ray Kremer did grow weary of reciting it.

Then the great sage said to himself, "Would it not be easier for me to write the sermons upon scrolls, and then when an apprentice comes who is in need of the wisdom of the sermons, he can read the scrolls, and I can save my voice." And so he did this, and set many of his sermons to page, and he built a library in which to house them, and this was the Library of Fak. And the sage Ray Kremer did delight much in directing the apprentices to go to the library and read the scroll which held the knowledge they did seek. Over the years, the library grew much, and it eventually held many times the scrolls that it had begun with. And the other sages of the Realm did praise the Library of Fak and the great sage Ray Kremer who had built it and filled it with scrolls.

Some time later, the great sage Doug Burkett found that he very much enjoyed the sermons given by the other sages, but he was saddened because many of the sermons were given but once, and they quickly faded from memory. And so Doug Burkett also began to set the sermons to page, his own and those of the other sages also. At first he stored his scrolls in an annex of the Library of Fak, which the great sage Ray Kremer was happy to provide. Later he took on an assistant, who built a separate library in which to keep the scrolls, and this was the Library of TypLyst. And this library too was praised by the other sages of the Realm, and they supported it by often bringing the great sage Doug Burkett to hear sermons that deserved entry into his library.

Thus the two libraries stood throughout the ages, and they became known as two of the great stores of knowledge of the known world.

## *Legends of the discussion groups: "The Roots of Power"*

by Ray "Homer he ain't" Kremer *[0:52, 12/31/01]*

The great leviathans of the TI empire, the 89, 92, and 92 Plus, were much beloved by the apprentices. Many of the apprentices desired the Roots of Power, but the leviathans only had the magik tokens for Roots of Two Powers, and the apprentices did covet Roots of Greater Power. And they did wail and gnash their teeth and cry out, "The smaller cousins of the leviathans do have a token for Eks Roots of Power, which can be used to find the Roots of Greater Power. Where is the token for Eks Roots of Power on the leviathans which are greater in every way?"

The sages of the land of 89 and 92+ delivered to the apprentices the sermon that the leviathans do not know the token of Eks Roots of Power, but that the Roots of Greater power could be found through the magik incantations of Reciprokal Exponents. And the sages did lament that the apprentices should know this sermon already, and too many times do apprentices come to hear the sermon of the Reciprokal Exponents. And the sages did once say, we should enscribe the sermon on a tablet, and place the tablet on the tower overlooking the Realm of the discussion groups, who many pass through to arrive at the land of 89 and 92+, and place torches on either side of the tablet so that all will read it and never again ask to hear the sermon spoken aloud.

From atop Mount Texas, the great creator Paul King heard the sages, and did create the tablet of Topik o' the Year, and he enscribed the incantations of Reciprokal Exponents upon it, and placed it on the tower for all to see. But the tablet was of little comfort, for still far too many apprentices came looking for the token of Eks Roots of Power upon the leviathans, and they did ignore the tablet unless the sages pointed it out to them. And down through the ages, no apprentices did cause the sages to shake their heads and rolls their eyes more than the ones seeking the Roots of Greater Power.

## *Lost Tales of TI Discusion Groups: Jason Adams, the Bezerker, the angry*

by Forgotten Scribe [10:41, 1/6/02]

Long ago, there was an alliance, the name now forgotten, forming about a programing group. An over zealous member posted multiple messages of anouncement in several different boards. The regulars grew a little irritated about hearing about it so much and finally a nameless bystander spoke up.

The story would have most likely ened there, but a very angry person, took forth to "defend free speech" as he put it and called for war.

A certain troll, Glass Lion, saw Jason as a feeble mind to manipulate and assumed the forms of another forgotten name and a certain hero and challenged the defender.

A great war of verbal obsenities waged on day and night.  Great floods flooded the land.  No one was safe from Jason.

Soon after, the fake hero challenged Jason, the real hero was attacked by Jason.  He quickly made it known it was not him and did not wish to fight him.

Soon after, the psychopath Adams appologised claiming only to seek protection of free speech.

The other trolls new however it would not be that dificult to get him to snap again and as this was an age of Flood and Spaming, Jason was an ideal target for there taunts.

Spam became even more prevalent under Jason.  This very much angered those high above on Mount Texas.  Day in, day out, cleansing of the  comunity was nessessary.

Finally, he actually did leave, although trolls and others still borrowed his name for the the notoritity.

Glass Lion proceeded to continue his search for the long gone Slayer.  While much rejoicing was there, the flooding continued off and on for many months to come.

### [1.19]  Kraftwerk's "Pocket Calculator" song

I have to include a bit about this song, because it may be the only one ever written about calculators. It certainly isn't specifically about the TI-89/92+, or even about graphing calculators in general. It's just a little calculator oddity.

Kraftwerk is a band of four Germans who released their first album in 1971. They were one of the first bands to compose songs almost entirely of synthesizer instrumentation. *Pocket Calculator* is from the *Computerworld* album, released in 1981 on the Elektra Entertainment label, then again in 1988. Their music is usually classifed as dance/electronica.

Some sites claim that this song was a popular hit - maybe in Europe or Germany? There is a lot of *Pocket Calculator* paraphernalia for sale on ebay: posters, t-shirts and various LP (vinyl) releases, and pressings in different colors.

You can learn more about Kraftwerk at:

http://pulse.towerrecords.com/contentStory.asp?contentId=736

and you can get a MIDI file of the song here:

http://pw1.netcom.com/~yebba/kraft.html

The song lyrics, which are undoubtedly copyrighted by Kraftwerk:

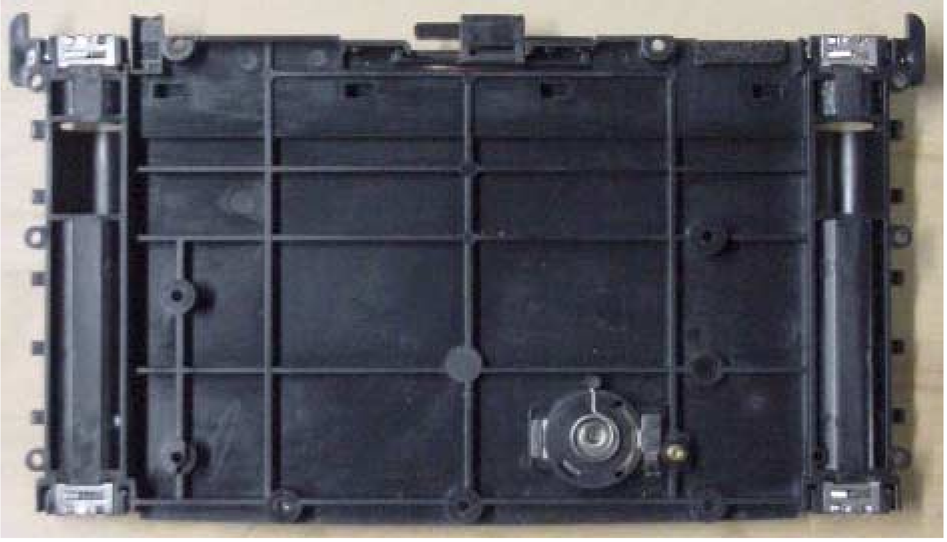
*I'm the operator with my pocket calculator*
*I'm the operator with my pocket calculator*

*I am adding and subtracting*
*I'm controlling and composing*
*I'm the operator with my pocket calculator*
*I'm the operator with my pocket calculator*
*I am adding and subtracting*
*I'm controlling and composing*
*By pressing down a special key, it plays a little melody*
*By pressing down a special key, it plays a little melody*
*I'm the operator with my pocket calculator*
*I'm the operator with my pocket calculator*

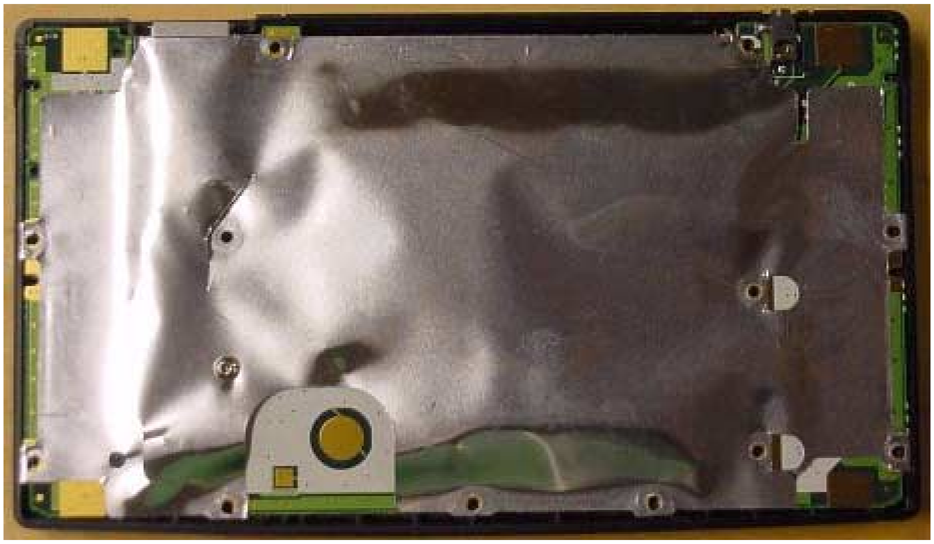
### [1.20] TI-92 Plus internal photographs

These photographs show the internal assembly of a TI-92 Plus, hardware version 2.

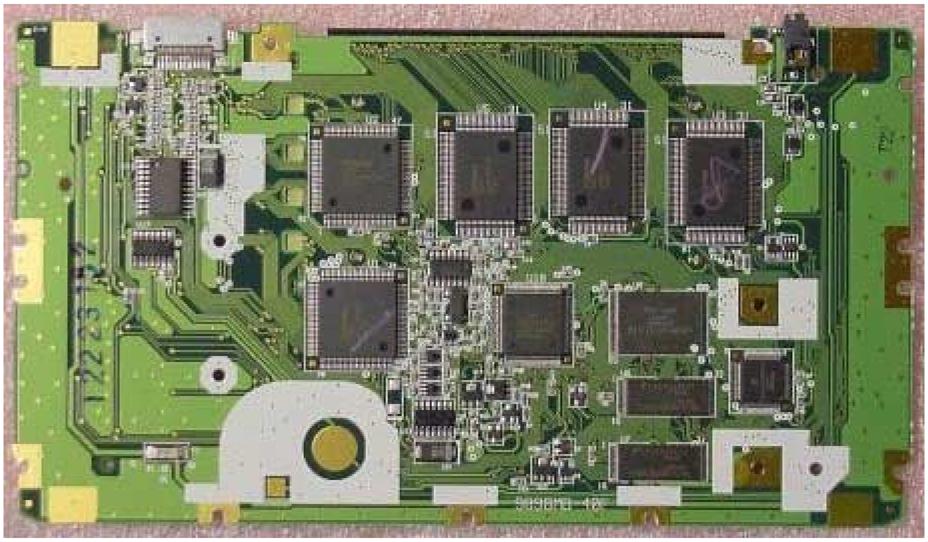
Inside the back cover



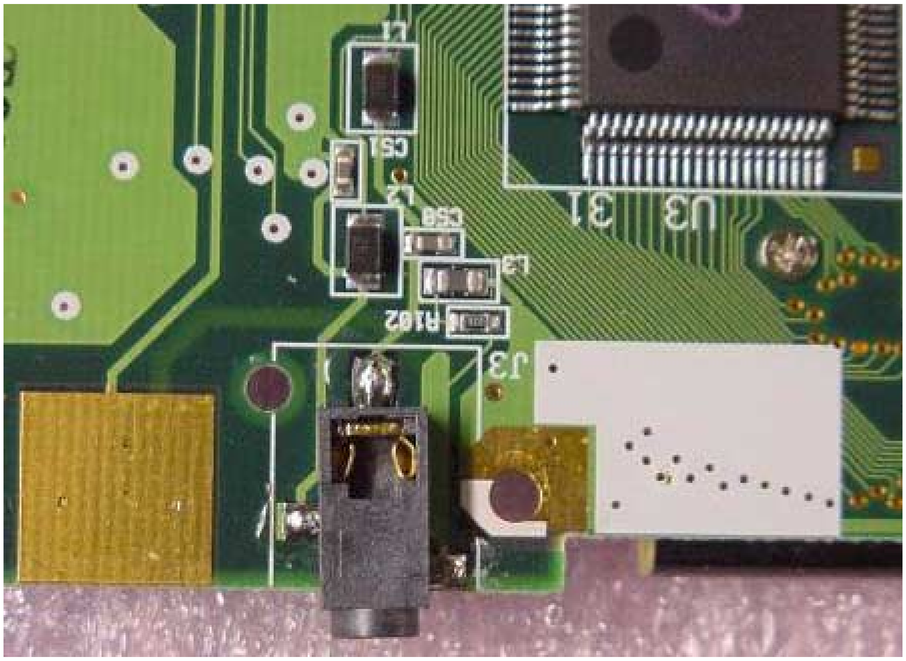
Component side of PCB with shield



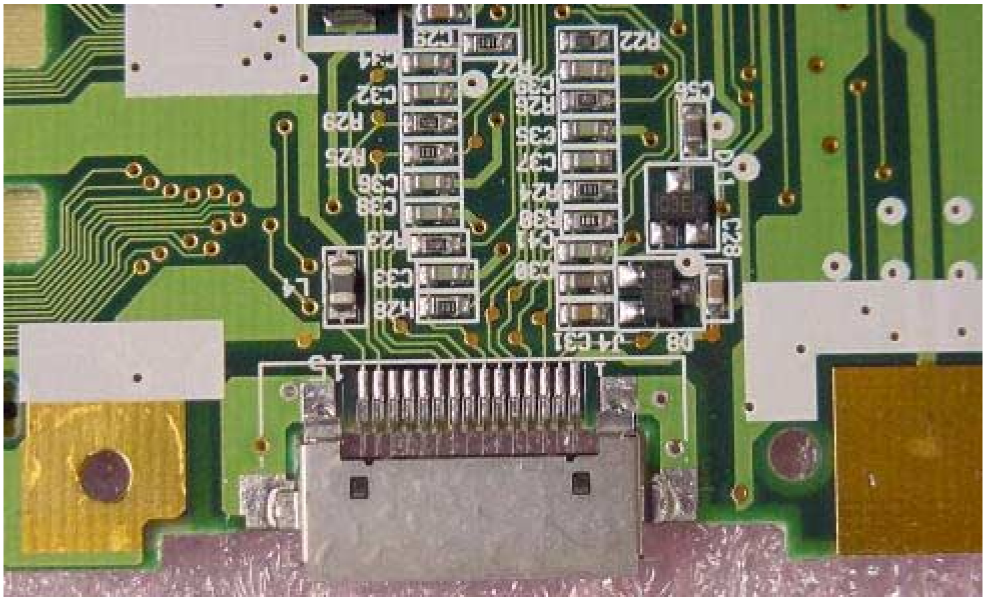
Component side of PCB



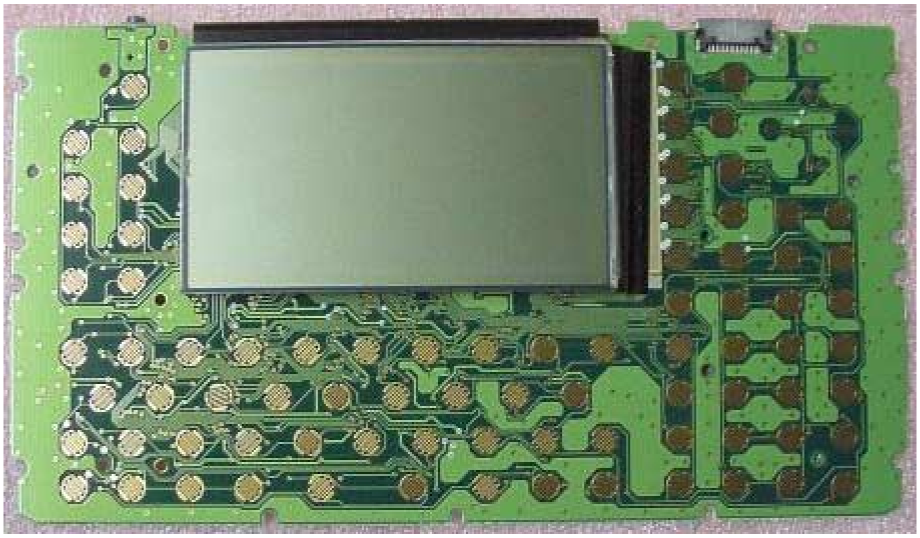
GraphLink I/O connector detail



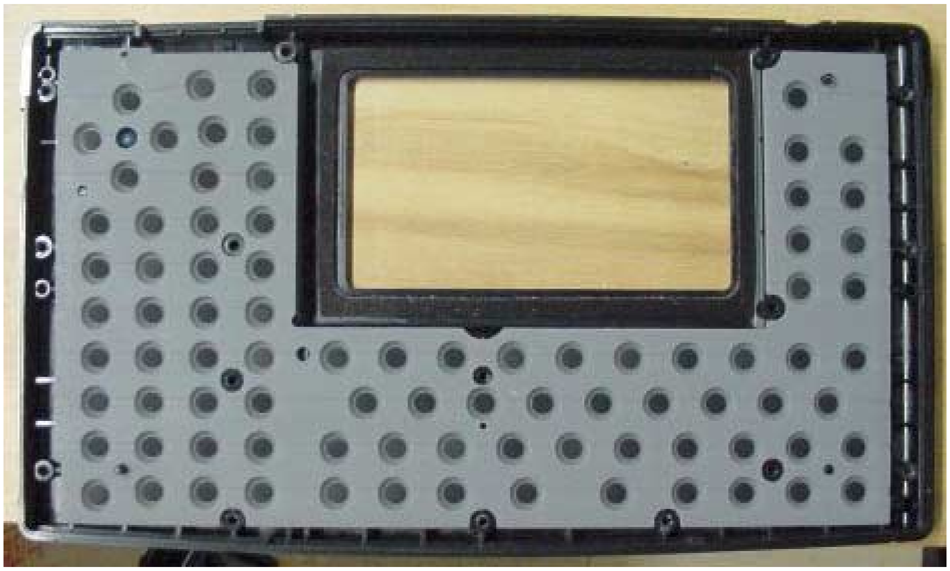
LCD connector detail



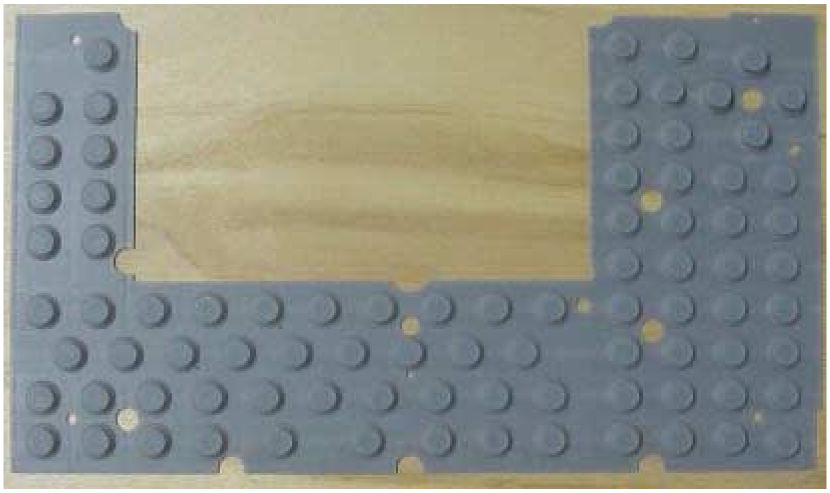
PCB, switch side



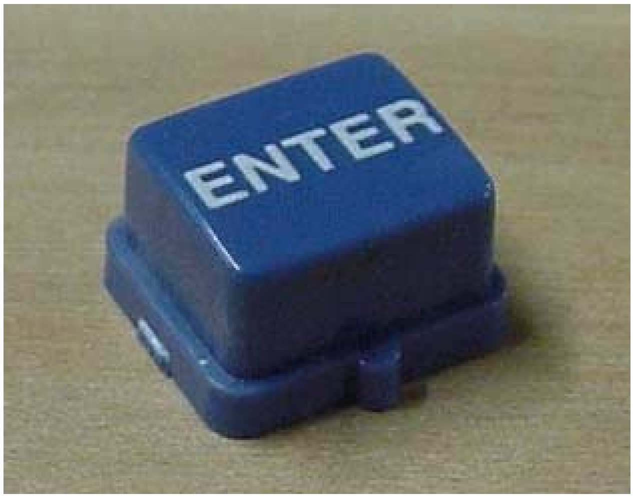
Key pad sheet, contact side



Key pad sheet, key side



Key cap detail

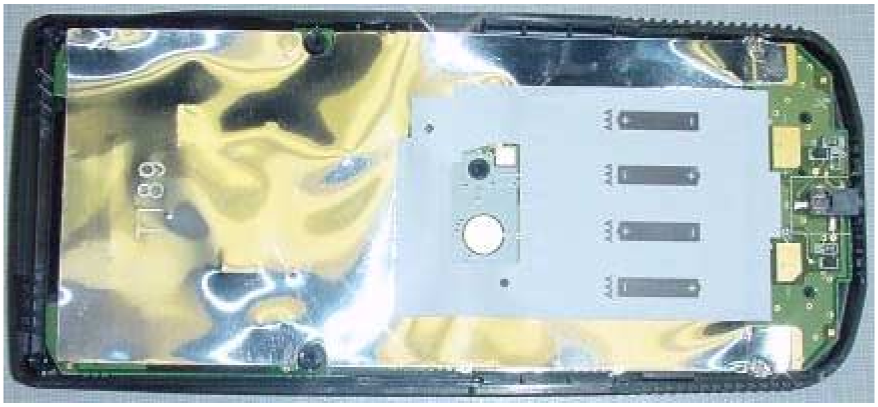


### [1.21]  TI-89 internal photographs

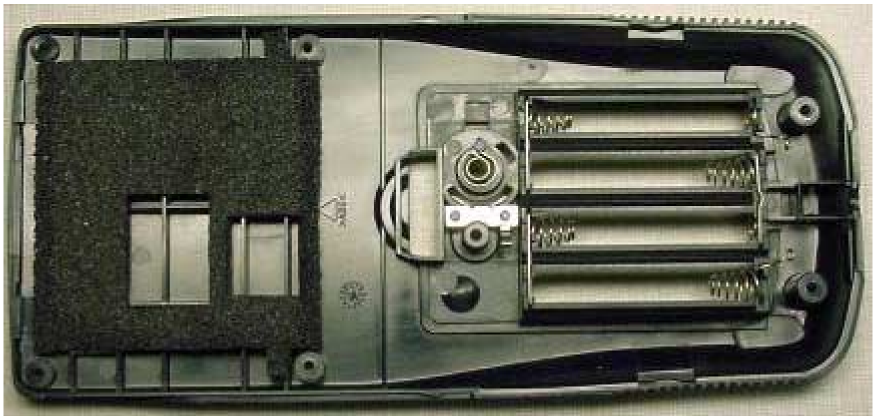
In case you're curious, the photographs in this tip show a disassembled TI-89, hardware version 1.

Back cover with batteries and battery covers removed

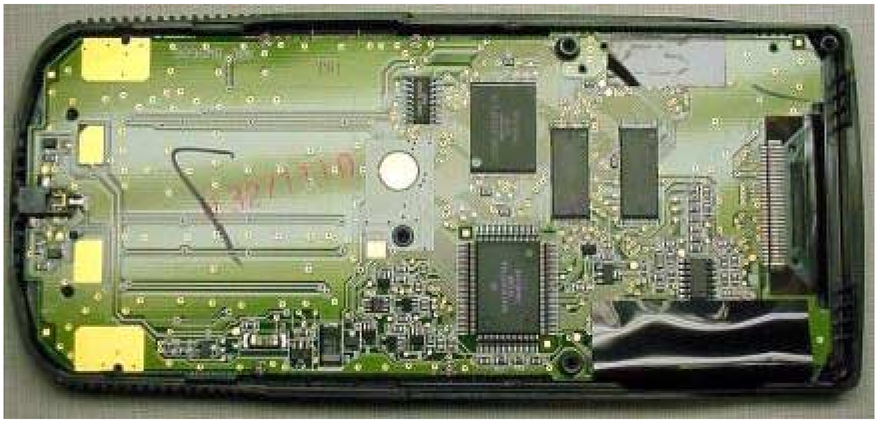
Back cover removed



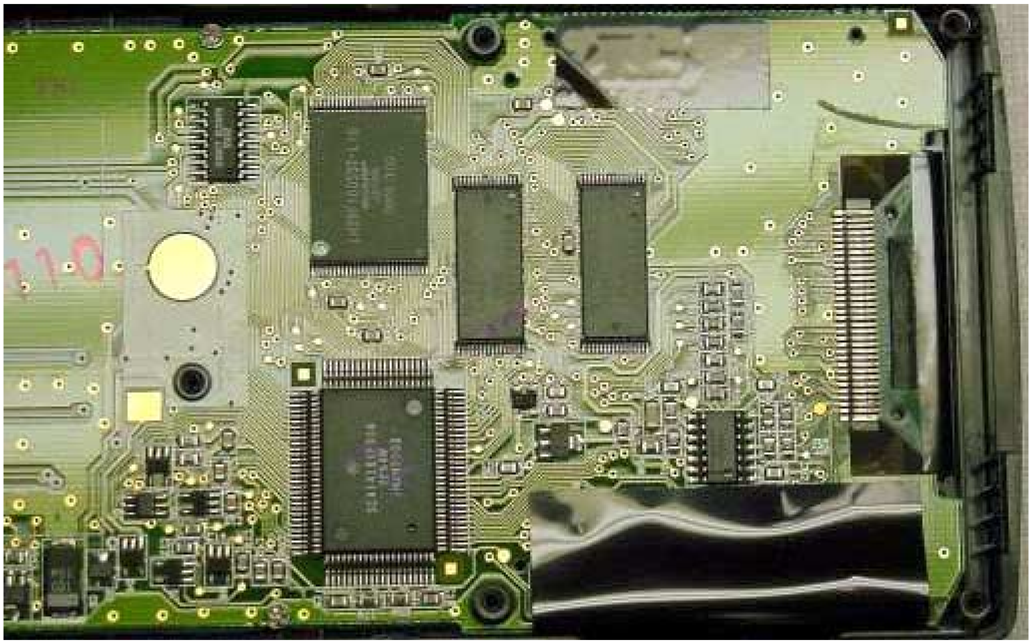
Inside view of back cover



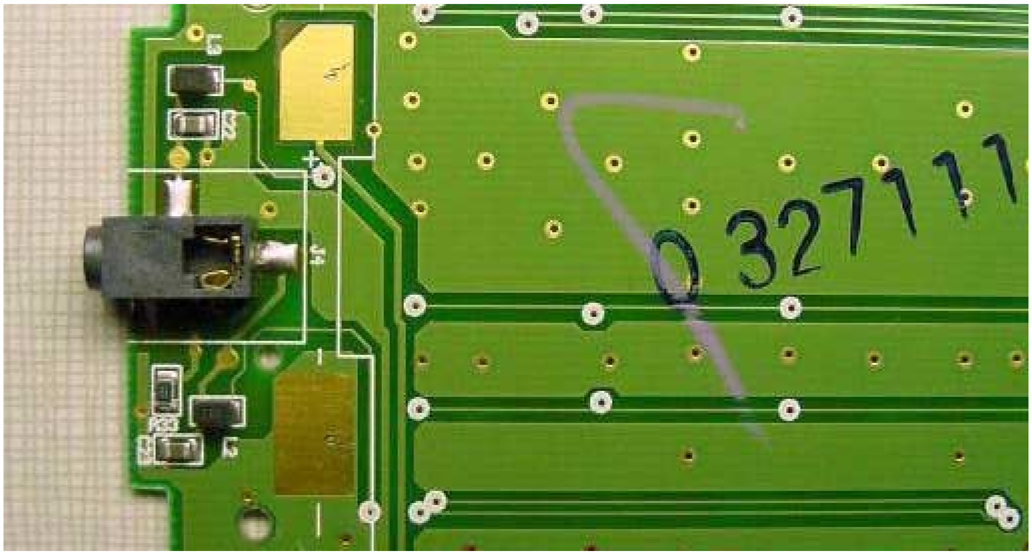
Component side of PCB with shield removed



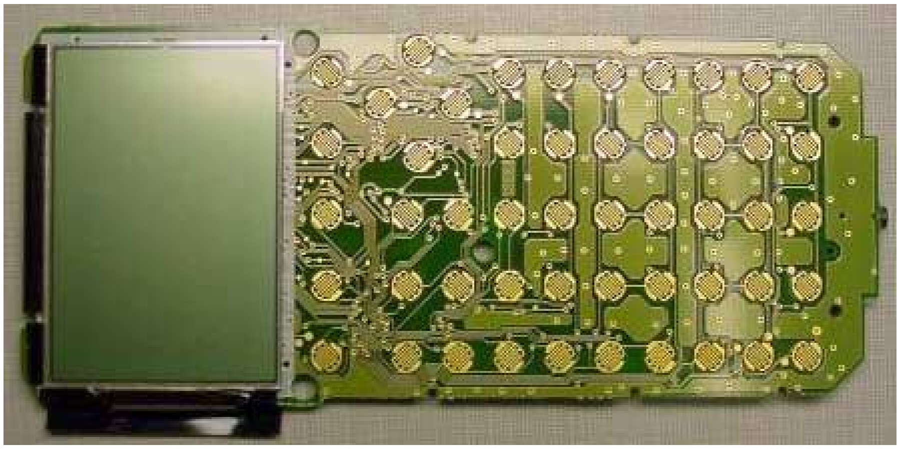
A detail view of the intergrated circuits



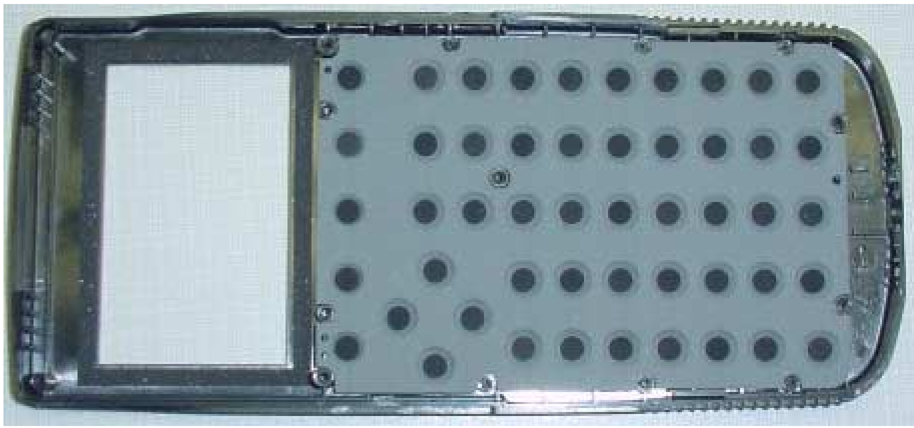
Detail view of the GraphLink I/O connector



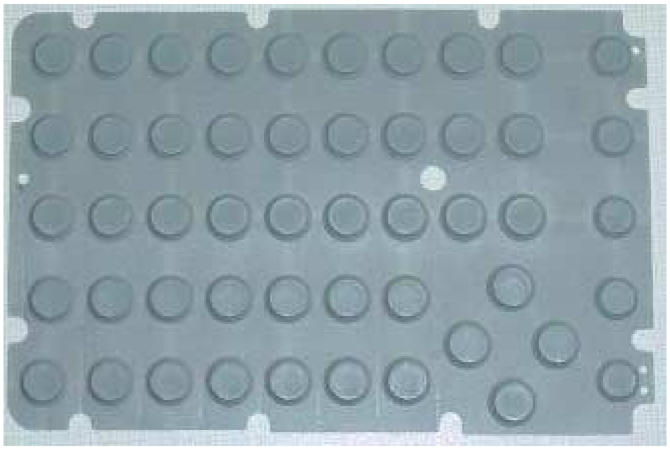
PCB, switch contact side



Case front cover with keypad rubber sheet in place



Keypad rubber sheet, key cap side



Inside front cover showing keycaps in place

Detail of a key cap

### [1.22] TI Calc-Con '02 Trip Report

*(In the spring of 2002, Ray Kremer, Bhuvanesh Bhatt and I met in Indiana, just to get together. Ray started calling this "Calc-Con '02", obviously as a joke. I posted the following report to the TI-89 / TI-92 Plus discussion group).*

I am happy to report that TI Calc-Con '02 was a success. No arrests were made and no emergency medical equipment was summoned. No live animals were used in the testing of various products. What follows is a more-or less chronological report ...

In attendance were Ray Kremer, Bhuvanesh Bhatt, myself, my wife, Janet, and my daughters Katie and Carrie. We arrived at the designated meeting place early, about 12:00, making much better time than we expected. The weather was cloudy and intermittently rainy for the trip from Eaton, but all in all, a nice drive. We had arranged to meet at the interpretive nature center at Fort Harrison State Park, which is just northeast of Indianapolis, Indiana.

Fort Harrison was actually a military facility from the 1950s, through the 1970s, and maybe earlier. However, it only became a state park in 1996, so they have a lot of work yet to do. Even so, this turned out to be a great place to meet for several different reasons.

As my family and I had a little extra time, we took advantage of the nature center. There were several interesting displays, including a diagram of a core sample of the earth beneath Indianapolis, down to 4000 feet, live exhibits of pond and creek fish, several species of turtles, and rat snakes, and microscopes with animal and botanical samples to inspect. There was also a tremendous display of about 200 animal skulls, and the labeled display called "The poop on scat" proved to be quite popular and informative.

Very shortly after 1:00PM, Ray and Bhuvanesh arrived. Both reported windy, rainy weather on the way in, but no major problems. As Bhuvanesh forgot about the time zone change, even with a reminder from Ray, he had to 'step on it' during the last leg of his trip. After introductions all around, and some general chatting, we revised our itinerary. The original plan was to explore Fort Harrison's hiking paths, then have a bit of a picnic, but unfortunately the weather did not cooperate. Continuous high winds were punctuated by dousing downpours, and the temperature was dropping rapidly. As none of us were outfitted for that kind of hiking, we repaired to the Fort Harrison restaurant for our midday repast, which was a quite satisfactory buffet. Lunch was quite enjoyable, as we talked about Ray's research project with copper tri-crystals, as well as Bhuvanesh's work at Wolfram. My wife acted as photographer for all of Calc-Con '02, and started soon after our luncheon.

I have to confess that I felt a bit underdressed for Calc-Con '02, as Bhuvanesh wore a spectacular oversized T-shirt with colorful math expressions, and Ray wore a shirt with the periodic table printed on the back. This is, perhaps, why he occasionally deprecates the need for calculator-based periodic tables - he's always wearing one!

So, instead of hiking, we decided to visit the Children's Museum in Indianapolis, which was also in keeping with the general technical theme of the convention. We found the Children's Museum with little trouble, although I was leading the caravan and forced Ray and Bhuvanesh to run every yellow light between Fort Harrison and the museum. We arrived at the museum a little before 4:00PM, so, with a closing time of 5:00PM, we had to get right to it. We enjoyed the water lab display and the mirror maze, but even my daughters found the life-size doll-house display a little boring. Ray turned out to be particularly good at building the 8' cantilevered arch. We managed to get several other activities in before closing time, including observing the albino raccoon. As the museum closed, we watched the very elaborate water-clock in the lobby, which was at least two stories tall. Because of our late lunch, no one was particularly in the mood for supper, so we attended a movie showing at the CineDome at the Children's Museum. The CineDome has a huge semi-spherical screen, with a resulting 'immersive' experience. After a short promotional film for the state of Indiana, we all enjoyed the main feature on animal migrations.

The film ended at 6:30, at which point we returned to Fort Harrison, as my family and I had reserved a room at the Harrison House for the night. I checked in, then we all headed for the room. The Harrison House was once military housing for officers, with two small apartments sharing a large, comfortable common room. This was an ideal arrangement, as we spent the evening in the common room. After some confusion in getting a Papa John's pizza delivery set up, Ray taught us a dice game called Farkle. This is a reasonably complex game, for which he obviously made up most of the rules. Even so, it was plenty diverting until the pizzas arrived. After a delicious pizza supper, we were refreshed and invigorated, and the real work of the conference began.

Nah, not really. We did talk quite a bit about calculators in general, and TI calculators in specific, but it was mostly a review of old issues, including:

- the ti-calc CDROM fiasco,
- the flash app contest,
- Ray's appearance as 'feature' on the TI student web page,
- the annoying 24K asm limit,
- the violent-orange color theme of the TI student web pages,
- advantages and disadvantages of TIGCC vs. the TI SDK,
- the place of graphing calculators in education, both now and in the future,
- speculation about the Steen Machine,
- more speculation about Oliver Miclo's problems with TI-France,
- the enigma of E.W.'s true identity (yes, we could not escape this, even here), and
- the various cartoons on the TI student page.

I brought pretty much my entire calculator collection, so Ray and Bhuvanesh got a chance to see and play with many HP calculators, including the HP-49G. This prompted many interesting stories and observations from both Ray and Bhuvanesh, as the calculators reminded them both of various anecdotes.

After much more talk about calculators, math, science and technology, Ray and Bhuvanesh headed for home at about 2:00 AM. I was sad to see it end. The following day my family and I went to downtown Indianapolis. Since this was not part of Calc-Con '02, I won't go into any detail, except to say that future conventions should make the effort to visit the observation tower, in the Soldier's and Sailor's monument in the center of town. You can take an elevator nearly to the top of the tower, which results in a truly spectacular view of Indianapolis and a pretty big chunk of Indiana, too. Not for the claustrophobic, though.

I had a completely wonderful time, and I believe that Ray and Bhuvanesh did, too. It was really rewarding to meet Ray and Bhuvanesh in 'real life', and they're both just great guys. We plan to do it again next year, and it would be even better to have more people there. We may change the location so that more people can attend, and I would like to consider opening it up to users of any calculator brand, not just TI. I will also try to get a TI representative involved, although I have no idea in what capacity. I am open to suggestions on this.

*Ray added these comments in another post:*

[16:19, 3/11/02] Re: TI Calc-Con '02 trip report by Ray Kremer

Let's see here.  The museum also has a marvelous setup involving pool balls rolling around on a track, on the way from the top to the bottom they would activate various bells and gongs. One section requires someone to turn a knob that runs an Archimedes' screw in order to bring the balls to the top. Another impressive section dropped balls from the top onto a trampoline and back up into a basket. Doug won the Farkle game, thanks to a stunning two straights in a row (a straight occurs when you roll all six dice and get 1 through 6, it scores 1500 points in a game where the goal is to reach 10000 points).

Doug's older daughter had fun poking around with his various calculators, and when the HP49G came out the first thing she commented on was how funny the buttons are to press.

### [1.23] Ray Kremer Interview on TI Student Feature page

*(In February 2002, this interview with Ray Kremer appeared on the Features section of the TI Student web page.)*

*Ray Kremer, the Man Behind the "TI Calculator FAQ"*

If you have ever visited the TI Discussion Groups, you know who he is. But did you ever wonder who the guy behind the legend is? Now's your chance to find out. He's Ray Kremer, and he'll advise you to "read this:"

| | |
|---|---|
| Name: | Ray Kremer |
| Age: | 23 |
| Home: | Algonquin, IL |
| Current field of study: | Material Science (Master's Program), Purdue University |
| Field of undergraduate degree: | Chemistry, Bradley University, Peoria, IL |
| Motivation for choosing chemistry: | The family business, which involves selling chemicals to the metal finishing industry |
| Career plans: | Joining the family business full-time. |

TI CONNECTION:

*What was your first TI calculator?*
I got a TI-85 under the Christmas tree in 1994. I spent the rest of the school break reading the manual.

*Which TI products do you own?*
A TI-86 and a couple of TI GraphLink cables.

*What is your favorite TI product?*
The TI-86 of course, although the TI-89 and TI-92 Plus are nothing to shake a stick at.

*Within the TI community, what are you famous for?*
I have written a fairly comprehensive list of Frequently Asked Questions about TI's graphing calculators. Although for sheer name recognition in the general TI community, I think some of the more prolific game authors have me beat. In the TI Discussion Groups I'm also famous (or is that infamous?) for replying to questions with a URL to my FAQ website instead of a direct answer.

*When did you start visiting the TI Discussion Groups?*
It was probably the fall of 1996 when I first wandered in.

*What made you keep coming back?*
The first time I came to the Discussion Groups, I went into the TI-85 area and read a few posts just out of curiosity about what was there. They were mostly questions of course, and I discovered that I could answer most of them just by virtue of having thoroughly read the manual. I rather enjoy helping people out, and it's a shame for students to not get the most out of their TI calculators, so I quickly became a regular fixture in the Discussion Groups.

*How many hours do you spend on the Discussion Groups each week?*
An average day's run through the new posts takes about an hour to an hour and a half, so that probably adds up to nine or ten hours in a week's time. It varies by season; summer tends to be a slow time, and the first month or two of the school year is very busy.

*How much do you like the color orange?*
Quite a bit, actually.

*So what's your *favorite* color?*
Well, uh, orange. Hence the background color of my home page and the color of the lettering of the FAQ's title graphic.

JUST THE FAQs, MA'AM:

*How old is the TI Calculator FAQ?*
I first wrote it during the summer of 1997, and it went up on the web a few days after I arrived at college that fall. Since then the main body has about tripled in size, and it has gained about a half dozen satellite pages.

*How did you come up with the idea to start it?*
When I first got a TI GraphLink cable, it took some time and research to get it properly configured on my computer, and I'm a computer savvy person. Naturally lots of people also have trouble hooking up their TI GraphLink, and I was regularly writing page-long responses in the TI GraphLink Discussion Group detailing all the things one might need to check in order to get it running. It got to be quite bothersome to retype this from scratch every week, and it occurred to me that if I could get the whole thing put up in some permanent location, I could just send everybody there, and I would never have to type it again. So the "dirty little secret" is that the main goal of the FAQ is to help me avoid typing when I respond to calculator questions. However, the system also allows me to deliver detailed and well-researched answers instead of having to answer things off the top of my head.

*How did you get to be so knowledgeable about TI products?*
Reading the manuals is most of it, and being naturally good at math helps. After that, it was just a matter of poking through TI's website and other TI-related websites that I ran across. The information is all out there; knowing where to find it is the main stumbling block. I also keep my ear to the ground for new information. Several items in the FAQ began as insights or reports from others in the Discussion Groups, and two of the math techniques listed are straight from my calculus teachers. There's also a handful of little tricks I came up with.

*Do you call it the "F-A-Q" or the "fak?"*
I use "fak."

GENERAL INTEREST:

*What are your favorite TI-related websites?*
The first things that come to my mind when you ask for well-done, useful TI-related sites are Doug Burkett's TI-TipList, Techno-Plaza, and ti-cas.org. I also frequent several other TI-related forums – TI-Programmers is one of the better ones.

*What is your favorite time of day? I'm most active at night.*
The problem is that I don't get tired after only a normal day's worth of being awake, so I'm usually still up when the sane people are in bed sleeping.

*What are your hobbies (other than frequenting the TI Discussion Groups)?*
I'll probably surprise no one in saying that I'm a sci-fi and anime geek. I also enjoy a good LEGO® Technic set. And video games of course, although my collection is more modest than most.

*What is your favorite news source?*
Talk radio.

*What is the last book you read?*
I'm currently working my way through a rather large stack of Star Trek novels... Disregarding those, I read *Good Omens* by Neil Gaiman and Terry Pratchett during the holidays.

*What is the best movie you've seen in the last six months?*
I'd be remiss as a geek if I said anything other than *The Fellowship of the Ring*. *Harry Potter* was also excellent.

*Who are your role models or sources of inspiration?*
You can't grow up watching *Star Trek* and not be affected by that. If I were to wake up one morning with pointed ears, I probably wouldn't be too bothered by it.

*How did you feel when you were asked to be a Student Center "Feature of the Month," an honor normally reserved for inanimate objects like products and support programs?*
Flattered. Although I joke that my FAQ and I are a veritable cliché in the TI Discussion Groups, around the rest of the internet's TI community, it isn't anywhere near being a household name like the major game archive sites are. I feel like a local celebrity suddenly thrust into the national spotlight.

# 2.0 Computer Algebra System (CAS) tips

### [2.1] Force complete simplification with repeated evaluation

The 89/92+ CAS does not always completely simplify expressions. For example, if you enter this on the command line

```
y³|y=2·√(2)·r
```

this is returned

```
8·2^(3/2)·r³
```

which is correct but not completely simplified. If you paste this result into the command line and enter it, the fully simplified result is returned:

```
16*√(2)*r³
```

Another way to force the complete simplificaton seems to be to evalute the expression in a function, like this:

```
tt(ex)
func
ex
Endfunc
```

Executing this call

```
tt(y^3|y=2·√(2)·r)
```

returns the completely simplified result. Sam Jordan offers these additional comments:

> *I've noticed the same thing with other large equations. If you enter an expression on the command line, the TI-89 will automatically simplify it. If it is too complex, it goes as far as it can, and leave it in a bit simpler state, but doesn't finish it. If you only paste the partially simplified answer back into the input line, and hit [ENTER] again, it will finish the job. It seems to set a limit on the amount of work that it does on each pass through the simplification routine. Eventually you get to the point where re-entering the answer no longer changes the result, and you can assume that the simplification is complete.*

*(Credit to Roberto-Perez Franco and Sam Jordan)*

### [2.2] Try constraints or solve() in symbolic equality tests

With the 89/92+ CAS, you can set two expressions equal to each other, and the CAS can evaluate the equation and  return *true* or *false* indicating whether or not the two expressions are equivalent. If the CAS cannot determine if the two expression are equal, it returns the original equation.

When the CAS fails to resolve the equation to *true* or *false*, you can sometimes add domain constraints to enable the CAS to evaluate the equation. For example, consider

$$\frac{\sqrt{x}-1}{\sqrt{x}+1}=\frac{x-2\sqrt{x}+1}{x-1}$$

In this form, the CAS returns the original equation. If you add domain constraints on *x* like this

$$\frac{\sqrt{x}-1}{\sqrt{x}+1}=\frac{x-2\sqrt{x}+1}{x-1}|x=y^2 \text{ and } y>0$$

the CAS will return *true*. I believe that this effect occurs because the CAS does not always interpret the square root function as exponentiation to the power of 1/2; see [2.10] *CAS square root simplification limitations*.

If both expressions are fractions, you can try this function:

```
frctest(a,b)
func

expand(getnum(a))*expand(getdenom(b))=expand(getnum(b))*expand(getdenom(a))

Endfunc
```

This function cross-multiplies the expanded numerators and denominators of the equation expressions. *a* and *b* are the expressions on the left- and right-hand sides of the equality. For the example above,

```
frctest((√(x)-1)/(√(x)+1),(x-2*√(x)+1)/(x-1))
```

returns *true*.

Alternatively, try solving the expression for a variable in the expression. For example,

```
solve((√(x)-1)/(√(x)+1)=(x-2*√(x)+1)/(x-1),x)
```

returns *true*, indicating that the two expressions are equal.

*(Credit for domain constraints method to Glenn E. Fisher; solve() method to Bhuvanesh Bhatt)*

### [2.3] Use *when()* for absolute value in integrals

AMS 2.03 and later support the *when()* function in integration, so it can be used for numeric solutions of integrals. This is especially useful when the integrand includes expressions using the absolute value. For example, if you try to evaluate this integral

$$\int_{-2}^{3}\left|x^2-1\right|dx$$

with this command

```
∫(abs(x^2-1),x,-2,3)
```

the calculator is busy for several seconds, then returns the original expression. The integral can be evaluated, though, using *when()* to implement the absolute value function:

```
x^2-1→f(x)
∫(when(f(x)≥Ø,f(x),-f(x)),x,-2,3)
```

which eventually returns the approximate answer of 9.3333 3370 4891 4. The exact answer of 28/3 can be found by integrating x^2 - 1 over the three intervals [-2,-1], [-1,1] and [1,3], and summing the absolute values of the results. So, the approximate answer is good to about seven significant digits.

This method works only in Approx mode. In Exact mode, the method still returns the original integrand.

A more accurate Approx mode result can be obtained without using this method, by integrating over the three intervals listed above. In this case, the result is correct to all 14 significant digits.

*(Credit to TM)*

### [2.4] Use Exact mode to ensure Limit() success

This is mentioned in the manual, but it is worth repeating here, because you can get wrong results without warning. For example, consider this limit, expressed in two different ways:

```
limit((1+.Ø5*x/n)^n,n,∞)

limit((1+x/(2Ø*n))^n,n,∞)
```

In Exact mode, both of these limits result in e^(x/20), which is correct. In Approx or Auto mode, the second example works, but the first example returns zero.

*(Credit to Bhuvanesh Bhatt)*

### [2.5] Try manual simplification to improve symbolic integration

The CAS may fail to integrate for some expressions, but can succeed if the integrand is expressed differently. For example, the CAS cannot find a solution to this integral:

```
∫(√(1/(r^2-x^2)),x)
```

but the correct result is quickly returned for this version:

```
∫(1/√(r^2-x^2),x)
```

*(Credit to Bhuvanesh Bhatt)*

### [2.6] CAS uses common convention for integral of x^n

In general,

$$\int x^n dx = \frac{x^{n+1}}{n+1}$$

is true, except for n = -1, where the result is undefined. Actually,

$$\int x^{-1}dx = \ln(x)$$

which is also correctly returned by the CAS. This is not a concern if you are doing problems by hand, but if you are doing this type of symbolic integration in a program, it is worth checking for this condition and trapping it as an error, or using the correct solution. Both Mathematica and MathCad return the general result, too.
*(Credit to Bhuvanesh Bhatt)*

### [2.7] Use *when()* in integration in AMS 2.03

With AMS version 2.03 and later, you can use the *when()* function in integration, for example,

```
∫(when(x<1,Ø,1)x,Ø,2)
```

returns 1.

### [2.8] Use *expand()* with respect to variable for faster results

The *expand()* function may work faster and more completely if you expand with respect to a variable in the expression. For example, this takes over 11 minutes on my 92 w/Plus module AMS2.03:

```
expand(((3*a+2*b)/(2*a^2-5*b)*z-(7*a-2*b)/(3*a-b^2))^2)
```

and also warns that *Memory full, simplification might be incomplete*, but this takes less than 4 seconds:

```
expand(((3*a+2*b)/(2*a^2-5*b)*z-(7*a-2*b)/(3*a-b^2))^2,z)
```

and returns the same result, without the *Memory full* warning message. Note that the only difference is that the *z* variable argument has been added in the second example. The result is

$$\frac{(3a+2b)^2z^2}{(2a^2-5b)^2}-\frac{2(3a+2b)(7a-2b)z}{(3a-b^2)(2a^2-5b)}+\frac{(7a-2b)^2}{(3a-b^2)^2}$$

which is not completely expanded. If this result is again expanded:

```
expand((3*a+2*b)^2*z^2/(2*a^2-5*b)^2-2*(3*a+2*b)*(7*a-2*b)*z/((3*a-b^2)*(2*a^2-5
*b))+(7*a-2*b)^2/(3*a-b^2)^2)
```

the result (not shown here) is returned after about 5 seconds, completely expanded.

*(Credit to Bhuvanesh Bhatt)*

### [2.9] Find more symbolic integrals, faster, with Real mode and constraints

Finding symbolic integrals can be more successful when you use constraints and set the Complex Format to Real instead of Rectangular or Polar. This integral is a good example:

$$\int_0^\infty \left(t^2e^{-st}\right)dt$$

which is entered as

```
∫(t^2*ℯ^(-s*t),t,Ø,∞)
```

If the mode is set to Rectangular or Real, *undef* is returned. If we constrain the soluton for s>0, like this:

```
∫(t^2*ℯ^(-s*t),t,Ø,∞)|s>Ø
```

but leave the complex format set to rectangular, the 89/92+ is 'busy' for a long time, then returns the original integral. However, if we constrain the solution to s>0 *and* set the mode to Real, the calculator quickly returns the correct answer: $2/s^3$.

Here is another integral that is sensitive to mode settings:

$$\int_{-\infty}^{\infty} \frac{1}{4\pi(p^2+z^2)^{1.5}} dz = \frac{1}{2\pi p^2}$$

which is entered as

```
∫(1/(4*π*(p^2+z^2)^(1.5)),z,-∞,∞)
```

I get these results on my TI-92 w/Pluse module, AMS 2.03. Real and Rectangular are the Complex Format mode settings, and Exact and Approx are the Exact/Approx mode settings.

- Real, Exact: returns answer quickly
- Rectangular, Exact: returns answer, not as fast; warning message: "Memory full, some simplification might be incomplete"
- Real, Approx: can't find integral
- Rectangular, Approx: can't find integral

Mode settings of Real, Exact seem to be the best starting point for symbolic integration.

So the moral of the story is this: if the 89/92+ won't evaluate your integral, try various complex modes and constraints.

*(I lost my note for the credit on this one. Sorry - it's a good one!)*

### [2.10] CAS square root simplification limitations

The 89/92+ CAS (computer algebra system) rarely treats the square root operator as identical to raising the argument to the 1/2 power, because, in general, the two operations are not equivalent. For example,

$$x^{\frac{3}{2}} - \sqrt{x^3}$$

does not simplify to zero. However, the expression does simplify to zero if we restrict x, in other words, these expressions

$$x^{\frac{3}{2}} - \sqrt{x^3} \mid x > 0 \qquad \text{or} \qquad x^{\frac{3}{2}} - \sqrt{x^3} \mid x \geq 0$$

both correctly return zero.

If you write programs that manipulate symbolic expressions, you need to consider the domain of the variables. For example, for x < 0,

$$\left(x^{\frac{1}{2}}\right)^3 = -(-x)^{\frac{3}{2}} \cdot i \qquad \text{and} \qquad \left(x^3\right)^{\frac{1}{2}} = \sqrt{-x^3} \cdot i$$

If x = -1, then the first expression gives -i, and the second is i. Both expressions *do not* simplify to x^(3/2), because the identity (x^a)^b = x^(a*b) is not true for non-integer exponents *a* and *b*.

You also need to consider the CAS' behavior for x^n when x = 0. For example:

0^n | n>0 = 0
0^n | n=0 = 1, with the warning *0^0 replaced by 1*
0^n | n<0 = 0^*undef*

*(Credit to Carlos Becker for straightening me out on this)*

### [2.11] Force *getnum()* and *getdenom()* to return correct results with *part()*

The *part()* function is used to return the parts of an expression. In some cases it does not return the expected results. For example:

```
getnum(a/b)
```

return *a* as expected, and

```
getdenom(a/b)
```

returns *b*, as expected, and

```
part(tan⁻¹(a/b),1)
```

returns *a/b*, again, as expected. But

```
getnum(part(tan⁻¹(a/b),1))
```

returns *a/b*, and

```
getdenom(part(tan⁻¹(a/b),1))
```

returns 1.

While it can be argued that this is mathematically correct, it is hardly useful. To force *getnum()* and *getdenom()* to return the expected results, save the result of *part()* in a variable, the use the functions on that variable.

```
part(tan⁻¹(a/b),1)→temp
```

Then

```
getnum(temp)
```

returns *a*, and

```
getdenom(temp)
```

returns *b*.

*(Credit to Frank Westlake)*

### [2.12] Testing equivalence of symbolic expressions

The CAS (computer algebra system) of the 89/92+ does the best it can, but it can return expressions in unexpected forms. To test if two expressions *expr1* and *expr2* are equal, just try

expr1 - expr2

which will return 0 if the two expressions are equivalent, or

expr1 = expr2

which will return *true* if the expressions are equivalent.

These tests may fail even if the expressions are equivalent, if the CAS does not include the necessary identities. As a last resort, you can calculate a numeric result for the two expressions, substituting numbers for the variables. As an example, consider the trigonometric identity

$$\theta = \tan^{-1}\left(\frac{b}{a}\right) \quad \text{and} \quad \theta = \sin^{-1}\left(\frac{b}{\sqrt{a^2+b^2}}\right)$$

where *a* and *b* are the sides of a right triangle. The CAS cannot recognize that these two equations are equal, because, in general, the two arguments are *not* equal. However, executing

```
approx(tan⁻¹(b/a)-sin⁻¹(b/(√(a^2+b^2))))|a=2 and b=3
```

returns 0.0. This approach requires care, because it can indicate that expressions are equal when they are not, or vice versa. For example, round-off errors in evaluating the functions may return a result that is very small, but not zero. Further, you need to be careful in selecting the numbers to use for the variables. These guidelines help somewhat:

1. Don't use 0 or 1. These cause sum and product terms to drop out when they should not.
2. Use numbers that are relatively prime. This prevents equality of specific ratios.
3. Don't use pi, fractions of pi, or multiples of pi for trigonometric expressions in radians. These can cause trigonometric identities to return equal values when they are not, in general, equal. Similarly, don't use even multiples of 90° for trigonometric functions in Degree mode.
4. Don't use integers, for the same reasons as 1 and 2 above.
5. Try using random numbers. In this case the condition would be ...)|a=rand() and b=rand(). Using this method repeatedly with the example above, I got several results of 0, but also 40E-15, -40E-15, 10E-15, and so on. If you use random numbers, make sure the numbers are valid for the function arguments. See tip [6.20] for functions to generate random numbers in a specific range.
6. Repeat the calculation with different values.

7. Try plotting the difference of the two functions over the range of interest.

### [2.13] Try *comDenom()* to find limits

*(Note: AMS 2.05 fixes the specific example in this tip, but the general idea may still be useful for other limits. Thanks to John Gilbertson for pointing this out.)*

Sometimes the 89/92+ CAS cannot find a limit to an expression, because of the way the expression is structured. And, sometimes, *comDenom()* can restructure the expression so that the CAS can find the limit. For example, the CAS cannot find the limit as x approaches 0 for this expression:

$$\frac{\frac{1}{\sqrt{1+x}}-1}{x}$$

However, `comDenom((1/(√(1+x))-1)/x)`

returns

$$\frac{1-\sqrt{x+1}}{x\sqrt{x+1}}$$

and the *limit()* function can find the limit of this expression, like this:

```
limit((1-√(x+1))/(x*√(x+1)),x,0)  =  -1/2
```

*(Credit to Olivier Miclo)*

### [2.14] Define functions that CAS can manipulate

The CAS will not be able to manipulate functions that you define in the program editor, nor from the command line with *Define*. For example, if you create the cosecant function like this:

```
csc(x)
func
1/sin(x)
endfunc
```

the CAS will not be able to integrate or differentiate this function - it just returns the function. However, if you create the function at the command line, like this

```
1/sin(x)→csc(x)
```

then the CAS can integrate and differentiate the function.

You need not create these functions from the command line. You can create one or more functions in a program, like this:

```
makefunc()
Prgm
1/(sin(x))→csc(x)
1/(tan(x))→cot(x)
1/(cos(c))→sec(x)
EndPrgm
```

Also, it is possible to create these functions in the Program Editor. Start the editor to create a New function. In the New dialog box, set the Type to Function and enter a Variable name. Delete the Func and EndFunc lines. Finally, enter only these lines to define the function:

```
:sec(x)
:1/cos(x)
```

This will result in a function that can be manipulated by the CAS. You cannot use comments in these functions.

*(Credit to Andy)*

### [2.15] Use change of variable in limits of *when()* functions

The CAS may have problems evaluating limits of expressions involving *when()* functions. For example, this limit will not be evaluated:

```
limit(when(x=Ø,1,sin(x)/x),x,Ø)
```

If the limit is expressed as

```
limit(when(X=k,L,f(X)),X,Ø)
```

then substitute x-1 for X to obtain

```
limit(when(x-1=k,L,f(x-1)),x,1)
```

The example above becomes

```
limit(when(x-1=Ø,1,sin(x-1)/(x-1)),x,1)
```

which returns the correct limit.

*(Credit to Martin Daveluy)*

### [2.16] Find partial fractions with *expand()*

A proper fraction is the ratio of two polynomials such that the degree of the numerator is less than the denominator, otherwise the fraction is called improper. Partial fractions is a method to convert an improper fraction to a sum of proper fractions. While the 89/92+ do not have a specific 'partial fractions' function, *expand()* performs that operation, as noted in the 89/92+ manual.

Some examples:

| | |
|---|---|
| For | $\frac{x2+3x+5}{x+2}$ |
| use | `expand((x^2+3*x+5)/(x+2))` |
| which returns | $\frac{3}{x+2}+x+1$ |

For $\frac{x^4-x^3+8x^2-6x+7}{(x-1)(x^2+2)^2}$

use
```
expand((x^4-x^3+8*x^2-6*x+7)/((x-z)*(x^2+x)^2))
```

which returns $\frac{-1}{x^2+2}+\frac{3x}{(x^2+2)^2}-\frac{1}{(x^2+2)^2}+\frac{1}{x-1}$

### [2.17] Use a 'clean' or empty folder for CAS operations

Results from symbolic operations may not be correct if values are already defined for the variables in your expression. However, you might not want to delete the variables in the folder in which you are working.

One solution to this problem is to create an empty directory specifically for performing symbolic manipulations. If you define no variables in this folder, symbolic operations will always be performed assuming all the variables are undefined, which is usually what you want.

If you name this folder something like *aaa*, it will always be at the top of the list displayed by Current Folder in the Mode menus, and you can quickly switch to it.

To use the folder, make it current with the Mode key or *setfold()* command. Perform the CAS operations as needed.

### [2.18] Delay evaluation to simplify expressions

Sometimes the CAS will not simplify expressions because it assumes general conditions for the variables. As an example, suppose that we have the expression

$$t^n$$

and we want to extract the exponent *n*. We could try using the identity

$$n=\frac{\ln(t^n)}{\ln(t)}$$

but the CAS will not simplify the right-hand expression to *n*, because this expression is undefined for t=0, and because this identity is *not generally true* for t<0. So, if we constrain the expression for t>0, like this

```
ln(t^n)/(ln(t))|t>0
```

the CAS returns 'n' as expected. While this example had a fairly simple solution, you may want to use this technique for more complex expressions in which the constraints are not immediately obvious. In addition, you may need to apply the constraints sequentially to get the desired result. To continue with the same example, suppose that we want to use the constraints

```
t=p^2   and     n=q^2
```

This will result in the desired simplification, since

```
ln(t^n)/(ln(t))|t=p^2 and n=q^2
```

returns q^2, which is just *n*, as desired. But this needs one more substitution, q^2 = n, to complete the simplification, and this *does not* work:

```
(ln(t^n)/(ln(t))|t=p^2 and n=q^2)|q^2=n
```

and in fact returns *Memory error!*

To avoid this problem, Bhuvanesh Bhatt offers the following program:

```
delay(xpr,constr)
Func
©delay(xpr,constr) evaluates xpr and then imposes constraint
©Copyright Bhuvanesh Bhatt
©December 1999
if gettype(constr)≠"EXPR" and gettype(constr)≠"LIST":return "Error: argument"
if gettype(constr)="EXPR"
return xpr|constr
local i,tmp:1→i:while i≤dim(constr):constr[i]→tmp:xpr|tmp→xpr:i+1→i:endwhile:xpr
EndFunc
```

*xpr* is an expression to be evaluated, and may include constraints. *constr* may be either an expression or a list. If *constr* is a list of constraints, then each constraint is evaluated in the *while()* loop. The call to evaluate our example is

```
delay(ln(t^n)/ln(t)|t=p^2 and n=q^2,p^2=t and q^2=n)
```

which returns *n*.

Finally, note there is a simpler way to extract the exponent in this example, by using *part():*

```
part(t^n,2)
```

returns *n*. This works regardless of the complexity of *t* or *n*, for example,

```
part((a*t+2)^(3*sin(n),2)
```

returns 3*sin(n).

*(credit to Bhuvanesh Bhatt)*

### [2.19] Restrict function arguments to integers

To restrict function arguments to the integer domain, you can use either *int()* or @nx with the "With" operator "|". Consider

```
cos(2·π·n)
```

which is 1 for all integer *n*. However, the 89/92+ CAS returns the general result, since it cannot know that *n* is an integer. One way to force the arguments to be integers is

```
cos(2·π·n)|n=int(x)
```

which returns 1. Another method is to use the arbitrary integer variable @nx, where *x* is an integer from 0 to 255. This method looks like this for x=0:

```
cos(2·π·@nØ)
```

which also returns 1. To type the "@" character, use [DIAMOND] [STO] on the 89, or [2nd] [R] on the 92+.

*(Credit to Hank Wu and Fabrizio)*

### [2.20] *factor()* with respect to list elements

You can use *factor()* to simplify expressions which include list elements. To simplify this expression

```
a*d[1]+b*d[1]*d[2]+d[1]*d[2]*d[3]
```

factor with respect to the list *d*, like this

```
factor(a*d[1]+b*d[1]*d[2]+d[1]*d[2]*d[3],d)
```

which returns

```
d[1]*(d[2]*d[3]+d[2]*b+a)
```

This is not fully simplified, since d[2] is not completely factored out, so factor again with respect to d[2]:

```
factor(d[1]*(d[2]*d[3]+d[2]*b+a),d[2])
```

which returns the fully factored expression

```
(a+(b+d[3])*d[2])*d[1]
```

In this example, *a* and *b* are simple expressions, but the method also works if *a* and *b* are also lists.

*(Credit to Bhuvanesh Bhatt)*

### [2.21] Differentiation and integration fail with *part()*

The *part()* function is used to extract parts of an expression. However, in AMS 2.05, differentiation and integration can fail when applied to part. For example, this expression

```
d(part(e^(2*t)*sin(t),2),t)
```

returns 0, but it should return

```
2*(e^(2*t)
```

Incorrect results are also returned if you try to integrate the result of a *part()* operation. One work-around is to save the result of the *part()* operation, then take the derivative or integral of the saved result. This example saves the *part()* result to a variable called *xx*:

```
part(e^(2*t)*sin(t),2)→xx
d(xx,t)
```

The correct differentiation result is returned because the contents of *xx* are differentiated.

*(Bug found by ES)*

## [2.22] Methods for some CAS functions

The descriptions below are from a file on the TI web site at

*ftp://ftp.ti.com/pub/graph-ti/calc-apps/info/92algorithms.txt*

These descriptions probably apply to the original TI-92, and may not accurately describe the latest version of the 89/92+ CAS. Even so, they may be useful as a general indication of how the functions get their results.

*arcLen()*

arcLen(f(x),x,a,b) is merely $\int_a^b \sqrt{\left(\frac{d}{dx}(f(x))\right)^2+1}\,dx$

*avgRC()*

$$\text{avgRC}(f(x),x,h)=\frac{f(x+h)-f(x)}{h}$$

*cFactor()*
cFactor() is the same as factor() , except the domain is temporarily set to complex so that complex factors are sought and not rejected.

*comDenom()*
Term by term, comDenom(a/b+c/d) reduces (a*d+b*c)/(b*d) to lowest terms, and repeats this process for each additional term.

*cSolve()*
cSolve() is the same as solve() , except the domain is temporarily set to complex so that complex solutions are sought and not rejected.

*cZeros()*
cZeros(expr,var) is `expr▸list(cSolve(expr=Ø,var),var).`

*d() (symbolic differentiation)*
d() repeatedly uses the chain rule together with the well-known formulas for the derivatives of sums, products, sinusoids, etc.

*expand()*
Polynomial expansion is done by repeated application of the distributive law, interleaved with use of the associative and commutative laws to collect similar terms.

*factor()*
The methods include the well-known factorizations of binomials and quadratics, together with methods described by Geddes, Czapor and Labahn, *Algorithms for Computer Algebra*, Kluwer Academic Publishers, Boston, 1992. Laguerre's method is used for approximate polynomial factorization. (Press et. al: *Numerical Recipes* Cambridge University Press, 1986.)

*fMax()*
See fMin()

*fMin()*
For fMin(expr,var), if $d$(expr,var) cannot be computed symbolically, a combination golden-section parabolic interpolation search is performed for one local minimum. (R. Brent, *Algorithms for Minimization without Derivatives*, Prentice-Hall, 1973.) Otherwise, fMin(expr,var) is determined by solve(d(expr,var)=0,var), filtered by attempting to determine the signs of higher-order derivatives at these candidates. Surviving candidates are compared with the limits of *expr* as *var* approaches +infinity and -infinity; and also with points where *expr* or its derivative is discontinuous.

*integration (symbolic)*
Antiderivatives are determined by substititutions, integration by parts, and partial-fraction expansion, much as described by J. Moses (*Symbolic Integration: The Stormy Decade*, Communications of the ACM, August 1971, Volume 14, Number 8, pp. 548-559). Antiderivatives are NOT computed by any of the Risch-type algorithms. Definite integrals are determined by subdividing the interval at detected singularities, then for each interval, computing the difference of the limit of an antiderivative at the upper and lower integration limits. Except in exact mode, *nINT()* is used were applicable when symbolic methods don't succeed.

*limit()*
Limits of indeterminate forms are computed by series expansions, algebraic transformations and repeated use of L'Hopital's rule when all else fails. Limits of determinate forms are determined by substitution.

*nDeriv() (numeric derivative)*

$$\mathrm{nDeriv}(f(x), x, h) = \frac{f(x+h)-f(x-h)}{2h}$$

*nInt() (numeric integration)*
nInt() uses an adaptive 7-15 Gauss-Kronrod quadrature somewhat like the algorithm described in *Numerical Methods and Software*, by David Kahaner,Stephen Nash,Cleve B. Moler,George E. Forsythe and Michael A. Malcolm, Prentice Hall 1989.

*nSolve() (numeric solve)*
nSolve() uses a combination of bisection and the secant method, as described in Shampine and Allen's *Numerical Computing: An Introduction*, W.B. Saunders Co., 1973.

*Product function* Π() *(symbolic)*
Iterated multiplication is used for modest numeric limits.  Other products are determined by checking for telescoping products and simple patterns. Limits are used for infinite product limits.

*propFrac() (proper fraction)*
propFrac() uses polynomial division with remainder to determine the quotient polynomial and the numerator of the resulting fraction.

*solve() (symbolic solve)*
solve() uses factoring together with the inverses of functions such as ln(), sin(), etc.  Except in exact mode, these techniques might be supplemented by heuristics to isolate solutions, followed by nSolve() to refine approximations.

*Summation function* Σ() *(symbolic)*
Iterated addition is used for modest numeric limits.  Other sums are determined from formulas given in references such as "Standard Mathematical Tables and Formulae", CRC Press, together with tests for telescoping sums. Limits or special formulas are used for infinite summation limits.

*taylor() (taylor series)*
taylor(u,x,n,a) is Σ(limit(d(u,x,k),x,a)*(x-a)^k/k!,k,0,n), computed iteratively to avoid redundant computation of lower-order derivatives.

*tCollect() (trig collect)*
tCollect() repeatedly uses the well-known formulas for products and integer powers of sines and cosines, such as those given in the CRC*Standard Mathematical Tables and Formulae*.

*tExpand() (trig expand)*
tExpand() repeatedly uses the well-known formulas for the sine and cosine of multiple angles, angle sums and angle differences, such as those given in the CRC CRC*Standard Mathematical Tables and Formulae*.

*zeros()*
zeros(expr,var)  is  expr▸list(solve(expr=0,var),var)

### [2.23]  Try comDenom() for faster partial factoring

It is not be obvious that *comDenom()* can be used for factoring, but this is mentioned in the *TI-89/TI-92+ User's Guide,* in the *Alphabetical Listing of Operations* section. As an example, consider the expression

$6 \cdot x^5 - 8 \cdot x^4 + 2 \cdot x^3$ which can be factored as $2 \cdot x^3 \cdot (x-1) \cdot (3x-1)$

To partially factor an expression with *comDenom()*, include an argument variable which is not in the expression. I use *y* in this example:

```
comDenom(6*x^5-8*x^4+2*x^3,y)
```

which, in AUTO and EXACT modes, returns

$$2\cdot x^3\cdot\left(3\cdot x^2-4\cdot x+1\right)$$

In APPROX mode, a different but equivalent expression is returned:

$$6\cdot x^3\cdot\left(x^2-1.333\cdot x+0.333\right)$$

The *User's Guide* suggests defining a function to accomplish this:

```
comDenom(expr,exprvar)→comden(expr)
```

and adds:

> "Even when there is no denominator, the *comden* function is often a fast way to achieve partial factorization if *factor()* is too slow or if it exhausts memory.
>
> "Hint: Enter this *comden()* function definition and routinely try it as an alternative to *comDenom()* and *factor()*."

**[2.24] Infinity^0 evaluates to 1**

The TI-89/TI-92+ CAS evaluate infinity to the zero power as one. In general, x^0 is 1, for x not equal to zero, however, since infinity is not a number, the usual rules of arithmetic do not necessarily apply. Different CAS systems may return different results, for example, the HP-49G considers the result undefined or 1, depending on flag settings, and the HP-48, Mathematica and Maple all return 1.

One plausible rational for setting $\infty^0=1$ is that

$$\lim_{x\to 0}\left(\frac{1}{x}\right)^x=1$$

however, 1/x is not infinity, it is undefined.

There are actually different 'degrees' of infinity. A denumerably infinite set is equivalent to the set of all natural numbers. The German mathematician Georg Cantor defined an infinite set as one which can be put in one-to-one correspondence with a proper subset of itself. Cantor denoted the 'size' of a denumerably infinite set as $\aleph_0$ which is called aleph-null, aleph-zero, or aleph-naught. aleph is the first letter of the Hebrew alphabet. There are larger infinite sets, which are not in one-to-one correspondence with the set of natural numbers. These are designated $\aleph_1,\aleph_2,\aleph_3...$ and so on. Aleph-zero, aleph-one and so on, are called the cardinal numbers of infinite sets, where each set has a higher degree of infinity. These cardinal numbers are called transfinite numbers. If you are interested in learning more about transfinite numbers, I recommend chapter 7 of *Mathematics from the Birth of Numbers*, by Jan Gullberg, as an easy introduction. Unfortunately, Cantor's ideas were so radical that he encountered intense criticism from his peers, suffered from nervous breakdowns, and died in a mental institution. Cantor proved these results:

$$\aleph_0+\aleph_0=\aleph_0 \qquad \left(\aleph_0\right)^2=\aleph_0 \qquad 2^{\aleph_0}=\aleph_0^{\aleph_0}>\aleph_0$$

In any event, the TI-89/TI-92+ give this warning:

```
    Warning: ∞^Ø or undef^Ø replaced by 1
```

I emailed TI Cares about this operation. Here is the response:

*"An infinity symbol is necessary for doing limits, and if it is allowed as an argument or result of the limit function, then something must be done when it is combined with other expressions.*

*"What the TI-92 does in such compositions is consistent with extended analysis. It is also consistent with the ANSI standards for IEEE floating point arithmetic, and is the same as other CAS systems that treat infinity. For example,*

*inf - inf => undef*

*1/inf => 0*

*abs(1/0) => abs(+-inf) => inf*

*and*

*inf - 100 => inf*

*"For transformations such as the latter, it might help to think of inf as representing a whole set of numbers rather than a single one.*

*"In order to obtain sharp results in computer algebra, it is important to discard as little information as possible throughout each computation. For example, this is absolutely crucial for internal computations of limits, which is done recursively via rules such as the limit of an absolute value is the absolute value of the limit of the argument.*

*"It is true that at the more elementary levels of math education we lump more into the phrase "undefined", which therefore doesn't have a single definition. Rather, it means "I don't want to talk about that yet."*

*"However, for consistency, the computer algebra must implement one place along this spectrum of sophistication. Unless the place is "discard as little information as is practical", people will be disappointed that the product can't do certain limits, definite integrals, solutions of equations, etc. that are in standard math tables and relevant textbooks.*

*"Actually, we did weaken the product somewhat to appease the more elementary end of the spectrum: +-inf is necessarily carried internally, but it is degraded to undef during output. This is why abs(1/0) simplifies to inf but 1/0 is displayed as undef: Compositions can be more powerful than stepwise computations such as 1/0 STO foo: abs(foo). This is additional evidence that the only easily explained places on the spectrum are: a) No infinities: only undef as on typical purely numeric calculators; or b) Make it as powerful as is practical. The first choice is simply not an alternative if there is a desire to do symbolic limits, improper integrals, etc.*

*"I am sorry that a single product can't exactly match the needs of education from beginning algebra through second-year calculus. We chose to make the product as powerful as we could within the constraints on ROM and programming time. Many teachers have found that the excess power for some classes stimulates the students curiosity and provides opportunities for lively discussion. It also allows the students to buy a single calculator that suffices for a succession of courses."*

To which I would add this:

*A graduate student of Trinity*
*Computed the square of infinity.*
  *But it gave him the fidgets*
  *To put down the digits,*
*So he dropped math and took up divinity.*

- Anonymous

### [2.25] Factor on complex i to show polynomial complex coefficients

The CAS automatic simplification perturbs complex coefficients of polynomials, for example,

$$(a+bi)x^{2}+(c+di)x+e+fi \qquad \text{[1]}$$

becomes

$$ax^{2}+cx+e+\left(bx^{2}+dx+f\right)\cdot i$$

Use the *factor()* function to display coefficients associated with the independent variable, as in [1] above. For this example, use

```
factor(a*x^2+c*x+e+(b*x^2+d*x+f)*i,i)
```

where 'i' is the complex 'i', not the alphanumeric 'i'.

### [2.26] limit() fails with piecewise functions

The CAS has problems with limits of piece-wise functions defined with *when()*. For example,

```
limit(when(x=Ø,1,sin(x)/x),x,Ø)
```

returns itself, instead of the correct limit of 1. This problem occurs with limits of the form

```
limit(when(x=a,b,f(x)),x,Ø)
```

only when the first *when()* argument is an equality, and the complete limit at zero is taken. To get around this failure, use an independent variable of, for example, k-1, and evaluate the limit at 1. The original example becomes

```
limit(when(k-1=Ø,1,sin(k-1)/(k-1)),k,1)
```

which correctly returns 1.

*(Credit to Martin Daveluy)*

# 3.0 Data structure tips

## [3.1] Using indirection and *setFold()* with matrix elements

Suppose you have a matrix *M* of folder names, stored as strings, and you want to set the current folder in your program. This won't work:

```
setFold(#M[k,l])            (won't work!)
```

but this will:

```
setFold(#(M[k,l]))          (note extra parentheses around #M[k,l])
```

and so will this:

```
M[k,l]→fname
setfold(#fname)
```

*(credit to Jordan Clifford for extra-parentheses method)*

## [3.2] Calculate table indices instead of searching

Many functions require searching a table by an index value, then returning the corresponding value, or the two values that bracket an input value. Suppose we have this table:

| index | x | y |
|---|---|---|
| 1 | 0.5 | 0.11 |
| 2 | 0.7 | 0.22 |
| 3 | 0.9 | 0.33 |
| 4 | 1.1 | 0.44 |

If x = 0.7, the search routine should return 0.22. If x = 0.6, then the search routine should return 0.22, 0.33, or both, depending on the purpose of the program. For example, an interpolation routine would need 0.22 and 0.33.

In the most general case, the program searches the x-column data to bracket the desired x-value. Even for short tables and a fast search routine, this is slow. But if the x-data is evenly spaced and the first x-value is known, then it is faster just to calculate the index, like this:

$$\text{index} = \text{int}\left[\frac{x - x1}{dx} + 1\right]$$

where *x1* is the first x-value, *x* is the search target, and *dx* is the x-data spacing. For the table above, x1 = 0.5 and dx = 0.2. *int* is the 89/92+ *int()* function. To find the index for x = 0.8, the calculation is

$$\text{index} = \text{int}\left[\frac{0.8 - 0.5}{0.2} + 1\right] = 2$$

Note that this equation returns the index pointing to the x-value less than the target value.

If the x-data is in ascending order, *dx* is positive. If the x-data is in descending order, the formula works correctly if *dx* is negative.

Once you have found the index, you can find the corresponding table x-value from

$$x = dx(index - 1) + x1$$

If the target *x* is less than the first x-value in the table, the equation returns zero. If the target *x* is greater than on equal to the last x-value in the table, the equation returns an index larger than the table size. Your program should check for these conditions and handle them appropriately.

### [3.3] Delete matrix rows and columns

There is no built-in function to delete a matrix row. This will do it:

```
mrowdel(m,r)
Func
Return when(r=1 or r=rowDim(m),subMat(m,mod(r,rowDim(m))+1,1,rowDim(m)-1+mod(r,
rowDim(m))),augment(subMat(m,1,1,r-1); subMat(m,r+1,1)))
EndFunc
```

*m* is the matrix, and *r* is the number of the row to delete. You will get a *Domain error* message if *r* is less than or equal to zero, or greater than the number of rows of matrix *m*.

You can also use *mrowdel()* to delete a column, by finding the transpose, deleting the row, then finding the transpose again, like this:

```
(mrowdel(mᵀ,c))ᵀ→n
```

This deletes column *c* of matrix *m*.

*(credit declined)*

### [3.4] Reverse list elements

Use this function

```
listrev(l)
Func
©(list) reverse list elements
seq(l[i],i,dim(l),1,⁻1)
EndFunc
```

to reverse the elements of list *l*. For example, *listrev(*{1,2,3,4,5}) returns {5,4,3,2,1}

*(Credit declined)*

### [3.5] Unexpected *NewMat()* operation in Cylindrical and Spherical vector modes

If you use *NewMat*(1,2) to make a new matrix in the Rectangular vector format mode, you get [[0,0]] as expected. But if the vector format mode is set to Cylindrical or Spherical, you'll get

```
[[Ø,∠R▸Pθ(Ø,Ø)]]
```

While correct, this is hardly useful. In fact it will cause errors when used as an argument to programs that expect numeric matrix elements.

You will get similar results for sizes of (2,1), (1,3) and (3,1).

The work-around is to make sure you are in  rectangular Vector mode when you create the matrix.

This table summarizes the results of different Complex and Vector formats, and the resulting matrix that is created.

| Vector Format | Complex Format | newMat() argument | Result |
|---|---|---|---|
| RECTANGULAR | REAL, RECTANGULAR or POLAR | (1,2) | [[0,0]] |
| " | " | (2,1) | [[0][0]] |
| " | " | (1,3) | [[0,0,0]] |
| " | " | (3,1) | [[0][0][0]] |
| CYLINDRICAL | REAL, RECTANGULAR or POLAR | (1,2) | [[0,RP]] |
| " | " | (2,1) | [[0][RP]] |
| " | " | (1,3) | [[0,RP,0]] |
| " | " | (3,1) | [[0][RP][0]] |
| SPHERICAL | REAL | (1,2) | [[0,RP]] |
| " | " | (2,1) | [[0][RP]] |
| " | " | (1,3) | "Non-real result" |
| " | " | (3,1) | "Non-real result |
| " | RECTANGULAR or POLAR | (1,2) | [[0,RP]] |
| " | " | (2,1) | [[0][RP]] |
| " | " | (1,3) | [[0,RP,SPP]] |
| " | " | (3,1) | [[0],[RP],[SPP]] |

where RP is

```
∠R▸Pθ(Ø,Ø)
```

and SPH is

$$\angle \frac{\pi}{2} - \frac{\sin(\infty)\cdot\pi}{2} + \text{undef}\cdot i$$

and SPP is

$$\angle e^{\left[\frac{\text{sign}(0)\cdot\pi}{2} - \frac{\sin(\infty)\cdot\pi}{2}\right]\cdot i} \cdot \text{undef}$$

### [3.6] Convert True/False list to 0's and 1's

This tip demonstrates a combined use of the *seq()* and *when()* functions to convert a list of True and False elements to 1 and 0:

```
seq(when(list[x],1,Ø),x,1,Dim(list)))→list2
```

In this example *list[]* contains the True and False values, and *list2[]* contains the equivalent list of 0's and 1's.

*(Credit to Billy)*

### [3.7] Replacement for *Fill* instruction in functions

The *Fill* instruction is useful for building vectors and arrays of constants. However, since *Fill* is not a function, it cannot be used in 89/92+ functions. This can, though:

```
list▸mat(seq(val,i,1,nrow*ncol),ncol)
```

where *val* is the constant with which to fill the matrix, *nrow* is the number of rows, and *ncol* is the number of columns. For example

```
list▸mat(seq(2,i,1,6),3)
```

returns this matrix

$$\begin{bmatrix} 2 & 2 & 2 \\ 2 & 2 & 2 \end{bmatrix}$$

### [3.8] Work-around for writing to data variable elements

There are no built-in instructions to write to single elements in data variables. This can be limiting, since data variables are used in the statistics and regression functions. However, you can write to elements indirectly with this procedure:

```
datalist(adata)
prgm
©Convert data variable to matrix
©31dec99/dburkett@infinet.com
©adata: contains data variable name as string

local alist1,alist2,amatrix

©Convert data variable 'adata' to one list for each column
#adata[1]→alist1
#adata[2]→alist2

©Convert lists to matrix
augment(list▸mat(alist1,1),list▸mat(alist2,1))→amatrix

©Modify list or matrix elements as needed, for example:
3→amatrix[2,1]
4→amatrix[2,2]

©Convert lists to data variable
newdata #adata,amatrix

Endprgm
```

The basic idea is to convert the data variable to a matrix, change the elements as needed, then convert the matrix back to a data variable. This program assumes that the data variable has two columns with an equal number of rows. Since the program refers to the data variable, indirection is used (the # operator), and the data variable name is passed as a string. For example, to modify a data variabled named *tdata*, call *datalist()* like this:

```
datalist("tdata")
```

Note that the columns of data variables can be accessed with the column number in square brackets:

```
#adata[1]
```

returns the first column of the data variable named in *adata*. The column is returned as a list.

Note that this method will not work in a function, because the *newdata* instruction cannot be used in a function.

It is not necessary to convert the data variable to a matrix. You can just convert the variable to lists, as shown in the example, make the necessary changes to the list elements, then convert the lists back to a data variable like this:

```
NewData #adata,alist1,alist2
```

### [3.9] Replace matrix rows and columns

Sometimes you need to replace an entire matrix row with a different set of values. For example, suppose you have this matrix:

$$a = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

and you want to change the second row to [40, 50 ,60]. This is easily done with

```
[40,50,60]→a[2]
```

In this case, a[2] is interpreted as the second row of matrix *a*. Now the matrix looks like this:

$$a = \begin{bmatrix} 1 & 2 & 3 \\ 40 & 50 & 60 \\ 7 & 8 & 9 \end{bmatrix}$$

Unfortunately, replacing columns is not so easy. There is a method that will work in most cases. Suppose we want to change the second column in the original matrix to [20, 50, 80]. These steps accomplish the change:

```
aᵀ→b
[20,50,80]→b[2]
bᵀ→a
```

The basic idea is to use the matrix transpose operator, to convert the matrix columns to rows. Then, the new 'column' is inserted as a row. Finally, the transpose operator is used again to convert the matrix rows back to columns. Note that this will work even if the matrix elements are complex numbers.

*(Credit for row replacement method to Glenn Fisher)*

### [3.10] Appending is faster than *augment()* for adding elements to lists; *seq()* is even faster

Frank Westlake says that:

*When working with lists it can be much quicker to append rather than to augment new values. The append operation continues at the same rate, regardless of the size of the list, but the augment operation continues to decrease speed as the list grows in size. For example:*

```
apnd()
prgm
local i,list
{}→list
for i,1,100
 disp i
 i→list[i]
endfor
disp list
endprgm
```

*is much quicker than*

```
agmnt()
prgm
local i,list
{}→list
for i,1,100
 disp i
 augment(list,{i})→list
endfor
disp list
endprgm
```

The *apnd()* program executes in about 9.8 seconds, and the *agmnt()* program finishes in about 17.9 seconds.

However, if your list element expression is  simple enough to be used in the *seq()* function, this is much faster than either *augment()* or appending. This function:

```
seq(i,i,1,100)→list
```

executes in less than 2 seconds.

*(Credit to Frank Westlake)*

### [3.11] Store anything in a matrix

Matrices can hold more than just numbers. In fact, they can hold expressions, strings and variables. In most programs, this makes matrices a better data structure than a data variable, since you cannot directly write to the items in a data variable.

This tip from an anonymous poster: Lists can be saved in a matrix as strings. Use this to store the list:

```
string(list)→M[j,k]
```

where *list* is the list and *M* is the matrix. Use this to extract the list:

```
expr(M[j,k])→list
```

### [3.12] Don't use *Δlist()* function in data editor

*NOTE: this bug has been fixed in AMS 2.05. The Δlist() function can be used as a column function definition.*

In the Data/Matrix Editor, you can define functions in the header to create columns of entries. Refer to the 89/92+ manual, page 248, for details. AMS 2.03 has a new function, `Δlist()`; see manual page 463.

*Do not* use this function in a column header. An Internal Error will result, and you will not be able to modify or use the data variable, because *Internal Errors* will be the response to just about all actions on the data variable. The only solution is to delete the data variable and start over again.

*(credit to Eric Kobrin)*

### [3.13] Find matrix minor and adjoint

The 89/92+ have no built-in functions to find the minor and adjoint of a matrix, but these are easily accomplished with the built-in functions. The minor of an n x n square matrix A = [$a_{ij}$] is the determinant of the matrix that results from deleting row *i* and column *j* of A. If the minor is

$$\left| M_{ij} \right|$$

then the cofactor of $a_{ij}$ is defined as

$$(-1)^{i+j} \left| M_{ij} \right| = \alpha_{ij}$$

and the adjoint of A is defined as

$$\mathrm{adj}(A) = \begin{bmatrix} \alpha_{11} & \alpha_{21} & \cdots & \alpha_{n1} \\ \alpha_{12} & \alpha_{22} & \cdots & \alpha_{n2} \\ \cdots & \cdots & \cdots & \cdots \\ \alpha_{1n} & \alpha_{2n} & \cdots & \alpha_{nn} \end{bmatrix} \tag*{[1]}$$

However, we need not find the minors and cofactors to find the adjoint, because of this identity:

$$\mathrm{adj}(A) = A^{-1} \cdot |A| \quad \text{if} \quad |A| \neq 0 \tag*{[2]}$$

It is faster to calculate the adjoint with this identity than by finding the minors and cofactors, but the identity is true only if the matrix is non-singular. However, a singular matrix also has an adjoint. So, an optimized adjoint function uses [1] if the matrix is singular, and [2] if not. This function finds the adjoint of a matrix:

```
adjoint(m)
Func
©(m) Return adjoint of square matrix m
local k,n,i,j,d
```

```
det(m)→d

if d≠0 or gettype(d)="EXPR" then

 m^(⁻1)*d→n

else
 rowdim(m)→k
 newmat(k,k)→n

 for i,1,k
  for j,1,k
   (⁻1)^(i+j)*det((mrowdel(mrowdel(m,i)ᵀ,j))ᵀ)→n[j,i]
  endfor
 endfor

endif

return n

EndFunc
```

*adjoint()* uses equation [2] if the matrix is non-singular, or if the matrix is symbolic. There is no error checking, and a *Dimension* error occurs if the matrix is not square.

The process of finding the matrix minor is built into *adjoint(),* but *mminor()* returns the minor, if needed:

```
mminor(m,r,c)
Func
©(m,r,c) Return first minor r,c of matrix m

return det((mrowdel(mrowdel(m,r)ᵀ,c))ᵀ)

EndFunc
```

*adjoint()* and *mminor()* call *mrowdel(),* which is described in tip [3.3].

To find the adjoint of $\begin{bmatrix} a & b \\ c & d \end{bmatrix}$

use this call:

```
adjoint([[a,b][c,d]]
```

which returns $\begin{bmatrix} d & -b \\ -c & a \end{bmatrix}$

To find the adjoint of $\begin{bmatrix} 1 & 2 & 3 \\ 1 & 3 & 4 \\ 1 & 4 & 3 \end{bmatrix}$

use this call

```
adjoint([[1,2,3][1,3,4][1,4,3]])
```

which returns $$\begin{bmatrix} -7 & 6 & -1 \\ 1 & 0 & -1 \\ 1 & -2 & 1 \end{bmatrix}$$

As an example of a singular matrix, find the adjoint of $$\begin{bmatrix} 1 & 2 & 3 \\ 0 & 1 & 2 \\ 0 & 0 & 0 \end{bmatrix}$$

with

```
adjoint([[1,2,3][Ø,1,2][Ø,Ø,Ø]])
```

which returns $$\begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & -2 \\ 0 & 0 & 1 \end{bmatrix}$$

*(Credit to Mike Roberts for pointing out equation [2])*

### [3.14] Randomize list elements

Some simulations require a list of random integers, in which each integer appears only once. This program creates such a list:

```
randlist(n1,n2)
Prgm
©Create list 'm' of sequential integers in random order, from n1 to n2
©Idea by David Lloyd

local n

seq(k,k,n1,n2)→m
seq(rand(),k,1,dim(m))→n
sorta n,m

EndPrgm
```

*n1* and *n2* are the starting and ending values for the list elements. The resulting list is stored in the list *m*. For example, the call

```
randlist(Ø,4)
```

might return this for *m*:

```
{4, 2, 0, 1, 3}
```

The first *seq()* function creates a list of sequential integers from *n1* to *n2*. The second *seq()* function creates list of random integers from 1 to the size of *m*. The *sorta* command sorts both lists, using the random list as the key, so that when the random list is sorted, the sequential list elements are randomized.

Ideally, this program would be written as a function to return the randomized list. Unfortunately, *sorta* is a command, not a function, and cannot be used in functions. The *sorta* arguments must be global variables, not local variables. 89/92+ functions cannot write to global variables, only local variables, so *sorta* will not work in a function.

None the less, this method is sufficiently short that it can be embedded into your programs. The necessary global variable can be deleted when your program exits.

*(Credit to Daniel Lloyd)*

### [3.15] Turn Pretty Print off to display commas in lists, matrices

If Pretty Print is turned On, list and vector elements are separated with spaces in the history display:

{1,2,3} is shown as {1 2 3}

When Pretty Print is turned off, the elements are separated with commas:

{1,2,3} is shown as {1,2,3}

*(Credit to Glenn E. Fisher)*

### [3.16] Use arrays with more than two dimensions

The 89/92+ can manipulate 1-dimensional data structures (lists and vectors), and 2-dimensional data structures (matrices and data variables). The number of dimensions is called the *rank*. Lists and vectors have a rank of 1. Matrices and data variables have a rank of 2. Matrices with rank greater than two are useful in many cases. For example, arrays of rank 3 can represent fields and other physical models. Arrays of rank 3 and 4 can be used to represent tensors.

This tip shows how to create and store arrays of any rank on the 89/92+, and how to store and recall elements of those arrays. It also shows how many built-in list functions can be used to manipulate high-rank arrays, and gives a program that finds the transpose of a rank-3 array.

I use the built-in list data structure to store the arrays, because a list can represent an array of any desired rank. First, consider a 3-dimensional array with dimensions {$d_1$, $d_2$, $d_3$}. An element's location in this array is defined as {$a_1$, $a_2$, $a_3$}, where

$1 \le a_1 \le d_1$ $\quad$ $1 \le a_2 \le d_2$ $\quad$ $1 \le a_3 \le d_3$

For example, a 2x3x4 matrix would have d = {2,3,4}, and an element in the matrix might be located at a = {2,1,3}.

To create an array *mat1* with all zero elements, use

```
newlist(d1*d2*d3)→mat1
```

For example, to create a 2 x 3 x 4 array, use

```
newlist(24)→mat1
```

The array elements are saved in the list such that the last index changes most rapidly as we move towards the end of the list. If we have a 2x2x2 array, and we label the elements as $e_{a1,a2,a3}$, then the list would look like this:

$\{e_{1,1,1}\ e_{1,1,2}\ e_{1,2,1}\ e_{1,2,2}\ e_{2,1,1}\ e_{2,1,2}\ e_{2,2,1}\ e_{2,2,2}\}$

For a particular element location $\{a_1, a_2, a_3\}$, we can find the corresponding list element index *k* from

$$k = a_3 + (a_2 - 1)d_3 + (a_1 - 1)d_2 d_3 \qquad [1]$$

This can be expanded and factored to eliminate one multiply, and access each variable only once, like this:

$$k = d_3(d_2(a_1 - 1) + a_2 - 1) + a_3 \qquad [2]$$

The routines to save and recall elements to a 3-dimensional array are called *m3sto()* and *m3rcl().* *m3sto()* looks like this:

```
m3sto(v,l,d,m)
Func
©(value,location{},dim{},matrix{})
©Store value in 3D matrix
©12octØØ dburkett@infinet.com

v→m[d[3]*(d[2]*(l[1]-1)+l[2]-1)+l[3]]
m

EndFunc
```

where

v	is the value to store in the array
l	is a 3-element list that specifies the element location as $\{a_1, a_2, a_3\}$
d	is a 3-element list that specifies the list dimensions as $\{d_1, d_2, d_3\}$
m	is the array list or the name of the array list

*m3sto()* returns the original array *m* with value *v* stored at location *l*.

For example, if we want to put 7 at location {1,2,3} in array *mat1*, which has dimensions 2 x 3 x 4, then use this:

```
m3sto(7,{1,2,3},{2,3,4},mat1)→mat1
```

To recall an array element, use *m3rcl():*

```
m3rcl(l,d,m)
Func
©(location{},dim{},matrix{})
©Recall element from 3D matrix
©12octØØ dburkett@infinet.com

m[d[3]*(d[2]*(l[1]-1)+l[2]-1)+l[3]]

EndFunc
```

where

l	is a 3-element list that specifies the element location as $\{a_1, a_2, a_3\}$

d is a 3-element list that specifies the list dimensions as {$d_1$, $d_2$, $d_3$}
m is the array list or the name of the array list

*m3rcl()* returns a scalar result. To get the element at location {1,2,3} in a 2 x 3 x 4 array *mat1*, and save it in the variable *var1*, use

```
m3rcl({1,2,3},{2,3,4},mat1)→var1
```

The index formula can be extended as needed for arrays of higher dimensions. For example, for a 4-dimensional array, the index formula is

$$k = a_4 + (a_3 - 1)d_4 + (a_2 - 1)d_3 d_4 + (a_1 - 1)d_2 d_3 d_4 \tag*{[3]}$$

This is expanded and factored to

$$k = d_4(d_3(d_2(a_1 - 1) + a_2 - 1) + a_3 - 1) + a_4 \tag*{[4]}$$

This index formula is coded in routines *m4sto()* and *m4rcl(),* which are not shown here but are included in the tip list code file *tlcode.zip*. The calling arguments are identical to *m3sto()* and *m3rcl(),* but note that the the location and dimension lists have dimension 4, since the routines work on a 4-dimensional array.

In general, for an n-dimensional array, you will sum *n* terms to find *k*. The terms are

$$\begin{aligned}
k = a_n + & \qquad\qquad \text{(1st term)} \qquad [5]\\
(a_{n-1} - 1)d_n + & \qquad\qquad \text{(2nd term)}\\
(a_{n-2} - 1)d_n d_{n-1} + & \qquad\qquad \text{(3rd term)}\\
(a_{n-3} - 1)d_n d_{n-1} d_{n-2} + & \qquad\qquad \text{(4th term)}\\
(a_{n-4} - 1)d_n d_{n-1} d_{n-2} d_{n-3} + \ldots & \qquad\qquad \text{(5th term)}
\end{aligned}$$

To find the simplified form of the sum for arrays with rank greater than 4, use equation [4] as a pattern. Begin by writing a nested product of the *d* terms, beginning with $d_n$, and stop when you reach $d_2$. I will use a 5-dimension array as an example, and the simplified form looks like this, so far:

$$k = d_5(d_4(d_3(d_2(\ldots$$

Next, start adding $a_k$ - 1 terms, and closing the parentheses as you go. Begin with $a_1$:

$$k = d_5(d_4(d_3(d_2(a_1 - 1) + \ldots$$

and continue until you reach $a_{n-1}$:

$$k = d_5(d_4(d_3(d_2(a_1 - 1) + a_2 - 1) + a_3 - 1) + a_4 - 1)\ldots$$

and finally add the $a_n$ term to the nested product:

$$k = d_5(d_4(d_3(d_2(a_1 - 1) + a_2 - 1) + a_3 - 1) + a_4 - 1) + a_5$$

This is the simplified form.

This function shows the general expression to find the list index *k* for an element location {$a_1$,$a_2$,...,$a_n$} for an array of dimension {$d_1$,$d_2$,...$d_n$}:

```
mrankni(a,d)
Func
©({loc},{dim}) return index
©Return list index for array element
©3ØnovØØ/dburkett@infinet.com; based on expression by Bhuvanesh Bhatt

Σ(when(i≠dim(a),(a[i]-1)*Π(d[j],j,i+1,dim(d)),a[i]),i,1,dim(a))

EndFunc
```

For example, to find the list index for the element at {1,2,3} in a 3x3x3 array, use

```
mrankni({1,2,3},{3,3,3})
```

which returns 6, meaning that the element at {1,2,3} is the 6th element in the list.

This function can be used for arrays of any rank, but it is much slower than the direct implementations given above. For a 4x4x4x4 array, the direct expression evaluates in about 0.037 seconds to find the list index *k*, while the general expression takes about 0.21 seconds.

Even though the general expression is slow, it is useful to find a symbolic expression for the list index. For example, to find the index expression for a rank-5 array, set

```
{a[1],a[2],a[3],a[4],a[5]}→a
{d[1],d[2],d[3],d[4],d[5]}→d
```

Then, executing *mrankni(a,d)* returns

```
a5+(a[4]+a[3]*d[4]+a[2]*d[3]*d[4]+a[1]*d[2]*d[3]*d[4]-
((d[2]+1)*d[3]+1)*d[4]-1)*d[5]
```

The expression is not optimized in this form but it can be optimized by factoring on *d* with *factor():*

```
factor(a5+(a[4]+a[3]*d[4]+a[2]*d[3]*d[4]+a[1]*d[2]*d[3]*d[4]-
((d[2]+1)*d[3]+1)*d[4]-1)*d[5],d)
```

which returns the optimized form

```
(((d[2]*(a[1]-1)+a[2]-1)*d[3]+a[3]-1)*d[4]+a[4]-1)*d[5]+a[5]
```

For some array operations, you may need to find the array element address {a}, given the list index. The three routines below implement this function. *m3aind()* finds the element address for a rank-3 array, and *m4aind()* finds the address for a rank-4 array. *mnaind()* finds the address for an array of any rank. It is slower than *m3aind()* and *m4aind(),* but obviously more general.

*Find element address for a rank-3 array:*

```
m3aind(i,d)
Func
©(index,{d1,d2,d3}) return {a1,a2,a3}
```

```
©Rank 3 matrix
©25octØØ dburkett@infinet.com

local a1,a2

intdiv(i-1,d[2]*d[3])+1→a1
intdiv(i-1-(a1-1)*d[2]*d[3],d[3])+1→a2
{a1,a2,i-d[3]*(d[2]*(a1-1)+a2-1)}

EndFunc
```

*Find element address for a rank-4 array:*

```
m4aind(i,d)
Func
©(index,{d1,d2,d3,d4}) return {a1,a2,a3,a4}
©Rank 4 matrix
©25octØØ dburkett@infinet.com

local a1,a2,a3

intdiv(i-1,d[2]*d[3]*d[4])+1→a1
intdiv(i-1-(a1-1)*d[2]*d[3]*d[4],d[3]*d[4])+1→a2
intdiv(i-1-((a1-1)*d[2]*d[3]*d[4]+(a2-1)*d[3]*d[4]),d[4])+1→a3
{a1,a2,a3,i-d[4]*(d[3]*(d[2]*(a1-1)+a2-1)+a3-1)}

EndFunc
```

*Find element address for array of any rank:*

```
mnaind(i,d)
Func
©mnaind(i,dims) returns the array location of ith list element
©Bhuvanesh Bhatt, Nov 2ØØØ

Local j,k,l,jj,a,dd

dim(d)→dd
1→d[dd+1]
1+dd→dd
true→jj

For l,1,dd
 d[l]→k
 If getType(k)≠"NUM" then
  false→jj
  Exit
 EndIf
EndFor

If getType(i)="NUM" and jj and i>product(d):Return {}

For l,1,dd-1
when(l≠1,intDiv(i-1-Σ((a[j]-1)*Π(d[jj],jj,j+1,dd),j,1,l-1),Π(d[k],k,l+1,dd)),int
Div(i-1,Π(d[jj],jj,2,dd)))+1→a[l]
EndFor

a
```

```
EndFunc
```

As an example, use *m3aind()* to find the element address of the 7th element of a 3x3x3 array:

```
m3aind(7,{3,3,3})          returns          {1,3,1}
```

This example for *m4aind()* finds the 27th element of a 4x4x4x4 array:

```
m4aind(27,{4,4,4,4})       returns          {1,2,3,3}
```

This example for *mnaind()* finds the address for the 40th element of a 5x4x3x2x1 array:

```
mnaind(40,{5,4,3,2,1})     returns          {2,3,2,2,1}
```

Note that *mnaind()* returns an empty list if an error condition occurs; your calling program can test for this condition to verify proper operation.

*mnaind()* is especially useful for finding the general expression for the addresses of elements for arrays of any rank. For example, use this call to *mnaind()* for a rank-5 array:

```
mnaind(k,{d[1],d[2],d[3],d[4],d[5]})
```

This will only work properly if *k* and the list *d* are not defined variables. The expression returned is quite lengthy (and not shown here), but it *is* correct.

The 89/92+ do not have built-in functions for higher-dimension arrays, but the simple list storage method means that simple array operations are trivial. These examples show operations on two arrays *m1* and *m2*, with the result in array *m3*. *k* is a constant or expression. The comment in parentheses shows the equivalent built-in 89/92+ array function. In general, *m1* and *m2* must have the same dimensions.

| | |
|---|---|
| Add two arrays (equivalent to .+) | `m1+m2→m3` |
| Add an expression to each element (equivalent to .+) | `k+m1→m3` |
| Subtract arrays (equivalent to .-) | `m1-m2→m3` |
| Subtract an expression from each element (equivalent to .-) | `m1-k→m3` |
| Multiply array elements (equivalent to .*) | `m1*m2→m3` |
| Divide array elements (equivalent to ./) | `m1/m2→m3` |
| Multiply array elements by an expression (equivalent to .*) | `k*m1→m3` |
| Divide expression by array elements (equivalent to ./) | `m1/k→m3` |
| Negate array elements | `-m1→m3` |
| Raise each array element to a power | `m1^k→m3` |
| Raise each *m1* element to *m2* power (.^) | `m1^m2 →m3` |
| Raise an expression to each element *m1* power (.^) | `k^m1→m3` |
| Take the reciprocal of each element | `1/m1→m3` |

| | |
|---|---|
| Find the factorial of each integer element | `m1!→m3` |
| Find the sine of each element<br>(also works for *cos(), ln(),* etc) | `sin(m1)→m3` |
| Differentiate each array element with respect to *x* | `d(m1,x)→m3` |

In general, more complex operations may be handled by nested for-loops that manipulate each array element. Or, in some cases, it is more efficient to process the list elements in sequence. The function below, *transpos(),* shows this approach.

```
transpos(arr,dims,i1,i2)
Func
©Example that transposes a rank-3 array on indices i1 and i2
©Bhuvanesh Bhatt (bbhatt1@towson.edu), Nov2000

Local i,tmp,arr2,dims2

If max(i1,i2)>dim(dims) or min(i1,i2)≤0:Return "Error: invalid indices"

newList(dim(arr))→arr2
listswap(dims,i1,i2)→dims2

For i,1,dim(arr)
m3aind(i,dims)→tmp
m3sto(m3rcl(tmp,dims,arr),listswap(tmp,i1,i2),dims2,arr2)→arr2
EndFor

arr2

EndFunc
```

Note that *transpos()* calls *m3aind(), m3rcl()* and *m3sto(),* as well as the *listswap()* function described in tip 3.18. These examples show typical results for *transpos().*

| | | |
|---|---|---|
| `transpos({a,b,c,d,e,f,g,h},{2,2,2},2,3)` | returns | `{a,c,b,d,e,g,f,h}` |
| `transpos({a,b,c,d,e,f,g,h},{2,2,2},1,2)` | returns | `{a,b,e,f,c,d,g,h}` |

*transpos()* is limited to rank-3 arrays, but it can be extended by changing the function references *m3aind()* and so on, as needed. Note that no error checking is done on the *dims* list or on the *i1* and *i2* input arguments, so make sure they are integers. As with a rank-2 array, the transpose function changes the dimensions of the array: if a 1x2x3 array is transposed on the second and third indices, the result is a 1x3x2 array. The *dims2* variable in *transpos()* above gives the dimensions of the resulting array.

If you are using Mathematica, note that regular cubic arrays in Mathematica can be converted to the storage format used in this tip, with Flatten[array].

Bhuvanesh Bhatt has coded quite a few tensor functions and programs which use the representation described in this tip. You can find them at his site, *http://triton.towson.edu/~bbhatt1/ti/,* in the *Arrays* package. These functions include:

*aInd(i,{dims})*
Returns the array location corresponding to the *i*th list index for an array of dimensions *dims*

*aRcl({loc},{dims},{arr})*
Returns the array element at location *loc* for an array *arr* of dimensions *dims*

*aSto(val,{loc},{dims},{arr})*
Returns the array *arr* of dimensions *dims* with the value *val* stored at location *loc*. To store this changed array, use `aSto(val,{loc},{dims},{arr})→myarray`

*newArray({dims})*
Returns the list corresponding to an array of dimensions *dims*

*Contract({arr1},{dims1},{arr2},{dims2})*
Contracts on the last index of *arr1* and the first index of *arr2*. To contract on other indices, first use *transpos()* to transpose the indices, contract, and then transpose the indices back.

*Diverg({arr},{dims},{coord},i)*
Returns the divergence of *arr* in the coordinates given by *coord* with respect to index *i*

*Gradient({arr},{dims},{coord})*
Returns the tensor gradient of *arr* in the coordinates given by *coord*

*Inner({arr},{dims},i1,i2)*
Performs an internal inner/dot product on indices *i1* and *i2* of *arr*

*Outer(f,{arr1},{dims1},{arr2},{dims2})*
Performs an outer/tensor/Kronecker product of *arr1* and *arr2*, using the function *f* to combine elements of *arr1* and *arr2*. *f* is usually "*".

*Transpos({arr},{dims},i1,i2)*
Transposes *arr* on indices *i1* and *i2*

*Christof({g},{coord})*
Returns the components of the Christoffel symbol of the 2nd kind for the metric *g* in coordinates *coord*

*Riemann({g},{coord})*
Returns the components of the Riemann tensor for the metric *g* in coordinates *coord*

*Ricci({g},{coord})*
Returns the components of the Ricci tensor for the metric *g* in coordinates *coord*

*RicciSc({g},{coord})*
returns the components of the Ricci scalar for the metric *g* in coordinates *coord*

*Einstein({g},{coord})*
Returns the components of the Einstein tensor for the metric *g* in coordinates *coord*

*arrays()*
Creates a custom menu

*aPrint(arr,dims)*
Displays nonzero components of the array *arr*

*delElem({list},i)*
Returns *list* with the *i*th element deleted

*listSwap({list},i1,i2)*
Swaps the *i1* and *i2* elements of *list*

*(Credit to Bhuvanesh Bhatt for the general index and address expressions and programs, the transpos() function, and lots of help with this tip!)*

## [3.17] Conditional tests on lists

Suppose you have two lists L1 and L2 and you want to know if they are identical, that is, each element of each list matches the other. The conditional '=' operator does not return a single *true* or *false* result when applied to the lists. Even so, this conditional expression will resolve to a single *true* or *false* result when used as the argument to an *If ... EndIf* or *when()* function. For example, this works:

```
If L1=L2 then
  (... code for L1 = L2 goes here ...)
else
  (... code for L1 ≠ L2 goes here ...)
endif
```

The other conditional operators also work for lists, for example:

L1 > L2 evaluates to *true* if each L1 element is greater than the corresponding L2 element
L1 < L2 evaluates to *true* if each L1 element is less than the corresponding L2 element
L1 ≠ L2 evaluates to *true* if each L1 element is not equal to the corresponding L2 element

*(Credit to Ray Kremer)*

## [3.18] Swap list elements

There is no built-in function to swap two list elements. This function exchanges the elements of *list* at locations *i1* and *i2:*

```
listswap(list,i1,i2)
func
©(list,i1,i2) list[i1]<->list[i2]
©Swap list elements
©29novØØ/dburkett@infinet.com
local x

list[i1]→x
list[i2]→list[i1]
x→list[i2]

list

Endfunc
```

For example,

```
listswap({a,b,c,d},1,4)
```

returns {d,b,c,a}.

This is a well known way to exchange two elements. I include it only because you might be tempted, as I was, to do it in a more convoluted way. Note that this function takes advantage of the fact that you can write to the function arguments. In this case, the elements of *list* are written directly to *list*, then *list* is returned.

This function executes in about 0.17 seconds on a TI92+ with AMS 2.05. For even better speed, don't call the function, just use the three lines that perform the swap. Those three lines execute in about 0.099 seconds.

### [3.19] Test a list for identical elements

If you need to know if all the elements of a list are identical, try this function:

```
lstident(l)
Func
©(list) return true if all elements of 'list' are the same
©18janØ1/dburkett@infinet.com
when(Δlist(l)=newlist(dim(l)-1),true,false)
EndFunc
```

The function returns *true* if all the list elements are equal, and *false* otherwise. The basic idea is to use `Δlist()` to create a list of the differences between the elements, then compare that result to a list of zeros.

This function works if the list elements are numbers (including complex numbers) or strings, but does *not* work if the list elements are expressions. If the elements are all the same expression, *lstident()* returns *true*, but if the elements are not the same, *lstident()* returns itself. It is a little odd that this method works for lists of strings, since *newlist()* returns a list with all zero elements, but

```
Δlist("a","a","a")          returns          {Ø,Ø}
```

and

```
Δlist("a","a","b")          returns          {Ø,"b-a"}
```

Since "b-a" is a string, and strings test as 'not equal' to numbers, it works.

Note that the *when()* function is necessary, otherwise the expression will return a list of *true* and *false* elements, instead of a single *true* or *false*.

### [3.20] Find matrix minimum and maximum

The built-in functions *min()* and *max()* return a row vector whose elements are the minimum and maximum, respectively, of the matrix rows. If you need the *single* minimum or maximum matrix element, use this to find the minimum matrix element of matrix *mat1*

```
min(mat▸list(mat1))
```

and this to find the maximum matrix element

```
max(mat▸list(mat1))
```

Both of these methods first convert the matrix to a list, then use *min()* and *max()* to find the most extreme element.

### [3.21] Convert matrices to single-row or single-column vectors

It is occasionally necessary to convert a matrix to a row- or column-vector. To convert a matrix to a column vector, use this:

```
mat2cvec(m)
Func
©([m]) convert matrix to column vector
©2april01/dburkett@infinet.com

list▸mat(mat▸list(m),1)

EndFunc
```

Or, to convert a matrix to a row vector, use this:

```
mat2rvec(m)
Func
©([m]) convert matrix to row vector
©2april01/dburkett@infinet.com

list▸mat(mat▸list(m),1)ᵀ

EndFunc
```

*mat2cvec()* converts a matrix to a row vector by first converting the matrix to a list, then converting that list to a matrix with one element in each row. *mat2rvec()* changes a matrix to a row vector in the same way, except that the column vector is transposed to make it a row vector. For example, if the input matrix is

$$m = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}$$

then *mat2rvec(m)* returns [a b c d e f g h i], and *mat2cvec(m)* returns

$$\begin{bmatrix} a \\ b \\ c \\ d \\ e \\ f \\ g \\ h \\ i \end{bmatrix}$$

### [3.22] Create lists with logarithmic element spacing

Lists with logarithmic element spacing are useful for creating function plots with logarithmic x-axis scaling, and for making tables of functions which change rapidly for small values of the indpendent variable, but less rapidly for larger values. This program creates such a list.

```
loglist(l,h,n)
Func
©(xl,xh,n) make list of n+1 elements from xl to xh with log-spacing
©19aprØØ/dburkett@infinet.com

local l1,d

log(l)→l1
(log(h)-l1)/n→d

seq(1Ø^(l1+k*d),k,Ø,n)

EndFunc
```

*loglist()* creates a list of n+1 elements which range from *l* to *h*. For example, if l = 20, h = 20,000 and n = 10, then the returned list is

{20, 39.91, 79.62, 158.87, 316.99, 632.46, 1261.9, 2517.9, 5023.8, 10023.7, 20000}

Note that the difference between the first two elements is about 20, and the difference between the last two elements is about 10,000. This is the desired effect of logarithmic spacing.

The function works by logarithmically mapping the original interval [l,h] onto the interval of [log(l), log(h)].

To make a logarithmic plot of some function f(x), use *loglist()* to make a list of elements over the required range of *x*, and save it as variable *list1*. Next, use

```
seq(k,k,Ø,n)→xlist
f(list1)→ylist
```

to make lists of x-coordinate plot values and function values, respectively. Finally, set up a data plot (not a function plot) in the Y= Editor, using *xlist* and *ylist* as the variables to plot.

### [3.23] Use single bits as flags to save status information

Sometimes a program needs to save and change the status of several conditions. For example, we may need to keep track of which operations the user has performed. If each of these operations requires a lengthy calculation and the user has already performed the operation, we want to skip the calculation. A typical method to accomplish this record-keeping is to use variables with boolean values *true* and *false*. This is effective if there are only a few status items we need to modify and record, but it becomes unwieldy if there are many items. If we had 30 status items, we would need to create and initialize 30 separate variables. Each boolean variable requires three bytes of RAM storage, so our 30 status items would use 90 bytes.

A variable that is used to indicate the status of a condition is called a *flag*. We can use RAM more efficiently if we save each status flag as a single bit in an integer. Since we will use the *and*, *or*, and *xor* operators to manipulate the flags, we can have, at most, 32 flags in a single integer. If you need more than 32 flags, you can use lists of integers. This method of using a single bit for a flag has these advantages, compared to using a single true/false variable for each status item:

- Less RAM is used, if the program requires many status flags.
- Large groups of flags can be quickly initialized.
- Fewer variable names may be needed, since each group of flags has only one name, and the bits are specified by a number.
- You can combine related status flags into single variables or lists, which makes it easier to manipulate them.
- You can quickly perform status tests on groups of flags with a *mask*.

There are four basic operations we need to perform on flags: set a flag, clear a flag, invert the status of a flag, and test to determine if a flag is set or cleared. While these operations can be performed most efficiently with C or assembly programs, it is possible to code these operations in TIBasic, as these four functions show:

| Function | Results |
|---|---|
| Set a bit: bitset(VarOrList,bit#) | Set bit# in VarOrList to 1 |
| Clear a bit: bitclr(VarOrList,bit#) | Clear bit# in VarOrList to 0 |
| Invert a bit: bitnot(VarOrList,bit#) | Invert bit# in VarOrList: if bit# is 0, set it to 1; if 1, set it to zero |
| Test a bit: bittst(VarOrList,bit#) | Return *true* if bit# in VarOrList is set, otherwise return *false*. |

For all four routines, *VarOrList* is a 32-bit integer variable or constant, or a list of 32-bit variables or constants. *bit#* is the number of the bit to manipulate or test. Bits are numbered starting with zero, so the bits in a 32-bit integer are numbered from zero to 31. Bit number 0 is the least significant bit, and bit number 31 is the most significant bit.

The 89/92+ use a prefix convention to specify 32-bit integers. The two prefixes are '0b' and '0h'. '0b' specifies a binary (base-2) integer consisting of digits 0 and 1. '0h' specifies a hexadecimal (base-16) integer consisting of digits 0 to 9 and A to F. If no prefix is used, the integer is represented as a base-10 number.

To demonstrate the functions, suppose our program needs to keep track of eight status flags, which we store in a variable called *status*. To clear all the flags, use

```
Ø→status
```

To set all the flags, use

```
ØhFF→status
```

Note that I use the *0h* prefix to indicate that *FF* is a hexadecimal (base-16) integer. You could also use

```
255→status
```

since, in binary (base-2) representation, both numbers are 0b11111111.

Suppose that all the flags are cleared, or set to zero. To set bit 0, use

```
bitset(status,Ø)→status
```

then *status* would be, in binary, 0b00000001.

Next, suppose that *status* is 0b00000111. To clear bit 1, use

```
bitclr(status,1)→status
```

then *status* would be 0b00000101.

We may need to reverse the status of a flag, and *bitnot()* is used to do that. Suppose that *status* is 0b00000000, then

```
bitnot(status,7)→status
```

results in *status* = 0b10000000, and bit 7 has been changed from 0 to 1.

Our program will need to decide what actions to take if certain flags are set or cleared. To perform a block of operations if flag 5 is set, use

```
if bittst(status,5) then
 ... {block} ...
endif
```

Or, to perform a block of operations if flag 7 is cleared, use

```
if not bittst(status,7) then
 ... {block} ...
endif
```

Note that the *not* operator is used to invert the result of *bittst()*.

All the examples shown so far have used a single 32-bit integer to hold the status flags. If you need more than 32 flags, you can use the same functions, but use a list of 32-bit integers instead of a single integer. For example, if we need 96 flags, we would use a list of three integers. The following example shows how to initialize such a list, then perform various operations on the flags in the list.

```
...
© Clear all the flags
{Ø,Ø,Ø}→statl
...
© Set bit 22
bitset(statl,22)→statl
...
© Clear bit 87
bitclr(statl,87)→statl
...
© Execute {block} if bit 17 is set
if bittst(statl,17) then
...{block}...
endif
```

For the purposes of the functions, you can consider the flag list to be a single integer which is 32*n bits, where *n* is the number of list elements. In reality, the flag bits are not numbered consecutively in the list. The bit numbers start at the least significant bit of the first element, and end at the most significant bit of the last element. For our example above, the bit numbers of the three integers are

{ |31|30|29|...|2|1|0|, |63|62|61|...|34|33|32|, |95|94|93|...|66|65|64| }

We can simultaneously test several status flags by using a mask. A mask is simply an integer with certain bits set or cleared, as needed to perform some desired operation. We will use the built-in boolean operators *and*, *or*, *xor* and *not* to perform mask operations. For example, suppose we want to execute a block of code if flags 0, 4 and 7 are all set. The mask to test these flags would be 0b10010001, and the code would look like this:

```
if status and 0b10010001=0b10010001 then
 ... {block} ...
endif
```

The result of and-ing the mask with *status* will only equal the mask if the mask bits are set. This method also provide more flexibility than the corresponding test with separate boolean variables, because the mask can be a variable, which can be changed depending on program requirements.

We can also use masks to perform other manipulations and tests. First, these truth tables show the operation of the built-in boolean operators.

| a | b | a and b |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

| a | b | a or b |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

| a | b | a xor b |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

You can use these truth tables to determine the mask values and boolean operators needed to perform different tests and manipulations, as shown:

| Description | Mask bit description | Operation | Example |
|---|---|---|---|
| Set a group of flags | 0: the corresponding flag bit is unchanged.<br>1: The corresponding flag bit is set. | *flags* or *mask* | Set flags 2 and 3:<br>*flags* = 0b0001<br>*mask* = 0b1100<br>result = 0b1101 |
| Clear a group of flags | 0: the corresponding flag bit is cleared.<br>1: the corresponding flag bit is unchanged | *flags* and *mask* | Clear flags 0 and 3:<br>*flags* = 0b0111<br>*mask* = 0b1001<br>result = 0b0001 |
| Invert a group of flags | 0: the corresponding flag bit is unchanged.<br>1: the corresponding flag bit is inverted | *flags* xor *mask* | Invert flags 1 and 2:<br>*flags* = 0b0011<br>*mask* = 0b0110<br>result = 0b0101 |
| Invert all the flags | (none) | not *flags* | *flags* = 0b1010<br>result = 0b0101 |
| Return *true* if all of a group of flags are set | 0: ignore the flag bit in the test.<br>1: include the flag bit in the test | *flags* and *mask* = *mask* | *flags* = 0b0111<br>*mask* = 0b0110<br>*flags* and *mask* = 0b0110<br>result = true |
| Return *true* if any of a group of flags are set | 0: ignore the flag bit in the test.<br>1: include the flag bit in the test | *flags* and *mask* ≠ 0 | *flags* = 0b0101<br>*mask* = 0b0110<br>*flags* and *mask* = 0b0100<br>result = true |

Masks can be used with lists of integers, as well as with individual integers. For example, suppose that *flags* = {0b0110,0b1100} and *mask* = {0b1111,0b1000}, then *flags* and *mask* = {0b0110,0b1000}

All the previous examples have shown the bit number argument passed as a number. It may, of course, be a variable instead, and you can improve program readability by choosing appropriate variable names for the flags. As an example, suppose that we have a program which may or may not calculate four sums, three limits and two integrals. We want to keep track of which operations have been done. We need a total of 9 flags, so we can save them all in one integer. At a slight expense in RAM usage, we can assign the flag numbers to variables like this:

```
Ø→sum1stat   ©Status flag names for sum1,
1→sum2stat   ©...sum2,
2→sum3stat   ©...sum3
3→sum4stat   ©...and sum4
4→lim1stat   ©Status flag names for limit1,
5→lim2stat   ©...limit2
6→lim3stat   ©...and limit 3
7→int1stat   ©Status flag names for integral1
8→int2stat   ©...and integral2
```

Now when we refer to one of the status flags, we use its name instead of the number, for example:

```
bitset(status,lim1stat)→status
```

would set flag 4 in *status*. To test whether or not the second integral has been calculated, we would use

```
if bittst(status,int2stat) then
 ... {block} ...
endif
```

All four of the bit manipulation functions do some simple error checking:

- The flags argument must be an integer or a list of integers. Any other type will return an error message.
- The bit number must be greater than -1, and one less than the total number of flags. If the flags argument is an integer, the bit number must be less than 32.

The same error message is used regardless of which error occurs. The error message is a string of the form "*name*() err", where *name* is the name of the function. Since one error message is used for all errors, there is some ambiguity as to which condition caused the error. This is not a serious flaw, because there are only two possible error causes. As usual, the calling routine can can use *gettype()* to determine the type of the returned result; if it is a string, then an error occurred.

Some possible errors are not detected. If the flags are passed as a list, the type of the list elements are not checked, that is, you must ensure that the list elements are integers. The functions do not determine if the argument is an integer, only that it is a number. Both integers and floating-point numbers have a type of "NUM", so the functons cannot distinguish between them.

A warning message is shown in the status line, when the bit number is 31 (for a single integer flag argument), or is the most significant bit of an integer in a list argument. The warning message is "Warning: operation requires and returns 32 bit value". This warning does not indicate a flaw in the functions, nor does it affect the operation.

The code for the four bit manipulation functions is shown below, and some explanation follows the listings.

Code for *bitset*() - set a bit

```
bitset(x,b)
Func
©(wordOrList,bit#) set bit#
©18mayØ1/dburkett@infinet.com

local k,msg,type
"bitset() err"→msg
gettype(x)→type

when(type="NUM",when(b<Ø or b>31,msg,exact(x or 2^b)),when(type="LIST",when(b<Ø
or b>32*dim(x)-1,msg,augment(left(x,k-1),augment({exact(x[k] or
2^(b-(k-1)*32))},right(x,dim(x)-k)))|k=1+intdiv(b,32)),msg))

EndFunc
```

Code for *bitclr*() - clear a bit

```
bitclr(x,b)
Func
©(wordOrList,bit#) clear bit#
©21mayØ1/dburkett@infinet.com

local k,msg,type

"bitclr() err"→msg
gettype(x)→type

when(type="NUM",when(b<Ø or b>31,msg,exact(x and not 2^b)),when(type="LIST",
when(b<Ø or b>32*dim(x)-1,msg,augment(left(x,k-1),augment({exact(x[k] and not
2^(b-(k-1)*32))},right(x,dim(x)-k)))|k=1+intdiv(b,32)),msg))

EndFunc
```

Code for *bitnot*() - invert a bit

```
bitnot(x,b)
Func
©(wordOrList,bit#) invert bit#
©21mayØ1/dburkett@infinet.com

local k,msg,type

"bitnot() err"→msg
gettype(x)→type

when(type="NUM",when(b<Ø or b>31,msg,exact(x xor 2^b)),when(type="LIST",when(b<Ø
or b>32*dim(x)-1,msg,augment(left(x,k-1),augment({exact(x[k] xor
2^(b-(k-1)*32))},right(x,dim(x)-k)))|k=1+intdiv(b,32)),msg))

EndFunc
```

Code for *bittst*() - test bit status

```
bittst(x,b)
```

```
Func
©(wordOrList,bit#) test bit#
©22mayØ1/dburkett@infinet.com

local k,msg,type
"bittst() err"→msg
gettype(x)→type

when(type="NUM",when(b<Ø or b>31,msg,exact((x and 2^b)≠Ø)),
when(type="LIST",when(b<Ø or b>32*dim(x)-1,msg,exact((x[k] and
2^(b-(k-1)*32))≠Ø)|k=1+intdiv(b,32)),msg))

EndFunc
```

All of the calculation for each function is performed with a single nested *when()* function: even though the listings above show three lines, each *when()* is a single TIBasic line. If the argument is a single integer, the function tests the bit number argument to ensure that it is between 0 and 31, since an integer has 31 bits. If the flags argument is a list, the functions test to ensure that the bit number argument is greater than zero, and less than or equal to the total number of bits less one, which is found by the expression 32*dim(x)-1.

For each function, a bit mask of the form $2^b$ is used to perform the required function. The bit mask is all zeros except for a 1 in the bit number argument position. For example, if the bit number is 4, then the bit mask is 0b10000. If the flags argument is a list, the bit mask is in the form $2^{b-(k-1)*32}$, which chooses the correct bit (from 0 to 31) in list element *k*. The list element *k* is found from the expression 1+intdiv(b,32), where *b* is the bit number input argument. In either case, the desired bit is set, cleared or inverted with a simple boolean expression:

*bitset()* uses x or $2^b$
*bitclr()* uses x and $2^b$
*bitnot()* uses x xor $2^b$

where *x* is the flags integer. From the boolean function truth tables shown above, you can verify that these operations result in the desired effect for the different functions. Note that each boolean operation is executed as the argument to *exact()*, which ensure proper results if the mode is set to Auto or Approx when the function is called.

If the flags argument is a list, then *bitset(), bitclr()* and *bitnot()* use a nested *augment()* function to assemble the original list elements, along with the changed element, into the returned list.

*bittst()* works a little differently, since it must return a *true* or *false* result. *bittst()* uses the boolean expression

```
exact((x and 2^b)≠Ø)
```

where *x* is the flags integer, and $2^b$ is the bit mask described above. This expression returns *true* if bit number *b* is set, otherwise it returns *false*.

### [3.24] Use equations to build lists of lists and matrices

It is usually not possible to create a list consisting of other lists. This returns an *Invalid list or matrix* error:

```
{3,{1,2},{3,4}}
```

and, interestingly, this

`{{1,2},{3,4}}` returns the matrix $\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$

However, you can create lists of lists and even more complicated data structures by using the list as an expression in an equation. Casting the first example in this form like this

```
{3,a={1,2},b={3,4}}
```

simply returns the list in the same form. Note that *a* and *b* must not exist as variables in the current folder. The second example above would be

```
{a={1,2},b={3,4}}
```

The equation variables *a* and *b* need not be unique. To reduce the chance that the equation variables are already in use, you can use international characters, for example

```
{ä={1,2},ä={3,4}}
```

On the other hand, you can use the equation variables to label the lists:

```
{result1={1,2},result2={3,4}}
```

You can build nested lists of lists:

```
{ä={1,2,ä={3,4,ä={5,6}}}}
```

To extract the individual list elements, use the *right()* function and the list element indices. Suppose we save a lists of lists to variable *r*:

```
{ä={1,2},ä={3,4}}→r
```

then

| | | |
|---|---|---|
| `r[1]` | returns | `ä={1,2}` |
| `r[2]` | returns | `ä={3,4}` |
| `right(r[1])` | returns | `{1,2}` |
| `right(r[2])` | returns | `{3,4}` |

You can also include matrices in lists with this method:

```
{ä=[1,2;3,4],ä=[5;6]}
```

*(Credit to Bhuvanesh Bhatt)*

### [3.25] Internal Error with seq() and Boolean operators in list index calculation

TI confirms that list index calculations involving a Boolean operator will fail with an Internal Error, when the calculation is performed in a *seq()* function call. This limitation applies to AMS 2.05. This will fail:

```
seq(t[1 and ØhF],k,Ø,1)
```

In this example, *t* is the list, and the Boolean *and* operator is used to calculate the index. *k* is the sequence variable, and its limits are immaterial. Remember that Boolean operations on integers must be executed in Exact or Auto mode; in Approx mode they will fail with a Data Type error.

The last example is the simplest possible command which demonstrates the error, but you would not likely attempt this, anyway. A more realistic example which fails is

```
seq(t[(shift(n,k*⁻4) and ØhF)],k,Ø,7)
```

Again, the offending party is the Boolean *and* operator. Any of the Boolean operators (*and*, *or*, *not*, *xor*) will cause the error. The index calculation itself does not cause the error, for example, this works:

```
Ø→k
t[(shift(n,k*⁻4) and ØhF)+1]
```

A Boolean operator will also cause an error if the summation function is used:

```
Σ(t[(shift(n,⁻4*k) and ØhF)+1],k,Ø,7)
```

There may be other built-in functions which are also susceptible to this error.

In most cases you can work around this limitation. I found the limitation while attempting this command:

```
sum(seq(t[(shift(n,k*⁻4) and ØhF)+1],k,Ø,7))→s
```

The equivalent work-around is

```
Ø→s
for k,1,8
 s+t[(n and ØhF)+1]→s
 shift(n,⁻4)→n
endfor
```

The work-around is effective because the index calculation is not embedded in a *seq()* function call.

I emailed TI Cares about this, and they claim it will be fixed in the next AMS revision.

### [3.26] Find indices of specific elements in lists and matrices

The built-in functions *min()* and *max()* return the minimum and maximum elements in lists and matrices, but there are no built-in functions that find the indices at which these values occur. This tip shows how to accomplish the more general task, of finding the locations of any target element, in TI Basic and in C.

*TI Basic version*

Considering lists first, the general problem is to find the indices at which a particular element is located, where that element is the minimum or maximum. This problem can be solved in a straight-forward way by comparing each element to the target value, as shown in this function:

```
indexlst(l,v)
func
©(l,v) return list of k|l[k]=v
©3nov01/dburkett@infinet.com

local k,r,j

{}→r                   ©Initialize result list ...
1→j                    ©... and result list index

for k,1,dim(l)         ©Loop to test each list element
 if l[k]=v then        ©If list element matches target ...
  k→r[j]               ©... save index in result list
  j+1→j                ©... and update result list index
 endif
endfor

return r               ©Return result list

Endfunc
```

Since more than one list element may match the target value, the matching indices are returned as a list. For example, if the list *mylist* is {5,4,7,3,2,1,1}, then *indexlst(mylist,5)* returns {1}, and *indexlst(mylist,1)* returns {6,7}.

*indexlst()* returns an empty list {} if the target element is not in the list. So, this function could also be used to determine if a list includes a particular element, by testing the returned result for *dim()* = 0.

This function can find the indices of the minimum and maximum list values, with the built-in *min()* and *max()* functions. To find the indices of the minimum list element, use

```
indexlst(mylist,min(mylist))
```

which, for the example above, returns {6,7}. To find the indices of the maximum list element, use

```
indexlst(mylist,max(mylist))
```

which returns {3}.

Finding the indices of a particular element in a matrix is only slightly more complicated, and we can use *indexlst()* to do the work, by converting the matrix to a list with `mat▸list().` Since *indexlst()* returns a list, we must then convert the list indices to the equivalent matrix indices. This function implements these ideas.

```
indexmat(m,v)
Func
©(mat,v) return matrix of [k,l]|mat[k,l]=v
©calls util\indexlst()
©4nov01/dburkett@infinet.com

local r,n,j,i,s,c

mat▸list(m)→r                  ©Convert the matrix to a list
util\indexlst(r,v)→r           ©Find list indices of v

if r={} then                   ©Return empty string if no matching indices
 return ""
else                           ©Convert list indices to matrix indices
 dim(r)→c                      ©... get number of matching indicex
 coldim(m)→n                   ©... get number of matrix columns
 newmat(c,2)→s                 ©... make result matrix
 for i,1,c                     ©... loop to convert indices
  intdiv(r[i]-1,n)+1→j         ©... find row index
  j→s[i,1]                 ©... save row index
  r[i]-n*(j-1)→s[i,2]      ©... find and save column index
 endfor
endif

return s                   ©Return matrix of indices

EndFunc
```

The function returns an empty string ("") if the target value is not found. The calling program or function can test the result with *getType()* to determine if there are any matching indices.

If the matrix *m* is
$$\begin{bmatrix} -2 & -2 & 8 & -7 \\ -5 & 9 & -7 & -5 \\ -2 & 0 & -3 & 9 \\ -8 & 0 & -5 & 9 \end{bmatrix}$$

then *indexmat(m,-2)* returns the result
$$\begin{bmatrix} 1 & 1 \\ 1 & 2 \\ 3 & 1 \end{bmatrix}$$

indicating that the element -2 is at indices [1,1], [1,2] and [3,1]. In the index matrix returned by *indexmat()*, the first column is the row indices of the original matrix, and the second column is the column indices. You can use *rowDim()* on the result matrix to determine the number of matching elements.

As with the list index function, we can use *indexmat()* to find the indices of the minimum and maximum matrix elements, but we must first use `mat▸list()` in the argument of *min()* and *max()*, because those functions return a row vector of column extrema for matrix arguments. So, to find the indices of the minimum elements of matrix *m*, we use

```
indexmat(m,min(mat▸list(m)))
```

which returns [4,1], since there is only one element which is the minimum value of -8.

To find the maximum element indices, use

```
indexmat(m,max(mat▸list(m)))
```

which returns
$$\begin{bmatrix} 2 & 2 \\ 3 & 4 \\ 4 & 4 \end{bmatrix}$$

and these are the indices of the maximum element 9.

*indexmat()* works on matrices of any row and column dimensions, and always returns a 2-column matrix. Note that you can use a single row-index value to extract the individual indices for the result matrix. For the example above, if we want the second index from the result, we use

[[2,2][3,4][4,4]][2]

which returns [3,4].

*C version*

The TI Basic functions above are functional but slow. The same functionality can be programmed in C, but the trade-off is increased hassle in that ASM functions, including those written in C, cannot be used in expressions. Tip [12.2] shows how to get around this by installing a couple of hacks. Note that you must use the correct version of *inlist()* for your calculator model: .89z for the TI-89, and .9xz for the TI-92 Plus.

The C version, *inlist()*, works the same as *indexlst()* above for lists. However, for matrices, *inlist()* is used to search for a row of elements, and not the individual elements.

An early version of inlist(), which only found a single element occurence, was about 22 times faster than the equivalent TI Basic version.

The source code for *inlist()* follows:

```
/*********************************************************************
This function searches a specified list for a specified item. If the
target item is found, the function returns a list containing the position
of item. If the item is found more than once, the function returns
a list containing all the indices of the target item. If the item is not
found, the function returns an empty list. No restrictions exist on the
types of items in the list.

Error Code Descriptions
90  -- Argument must be a list
930 -- Too few arguments
940 -- Too many arguments

Syntax: inlist(list, item)
        inlist(matrix, row)

Since a matrix is defined as a list of lists, this function will also
work with input such as inlist([1,2;2,3], {1,2})

Thanks to Samuel Stearley for his help in optimizing this program.

Brett Sauerwein
15 March 02
********************************************************************/

#define USE_TI89
#define USE_TI92PLUS
#define OPTIMIZE_ROM_CALLS

#define RETURN_VALUE

#include <tigcclib.h>

void _main(void)
{
    unsigned char *listPointer, *itemPointer;

    int argumentNumber = 0;
    int index = 1;

    // get a pointer to the first argument entered by the user

    InitArgPtr (listPointer);

    argumentNumber = remaining_element_count (listPointer);
    // throw the appropriate error if the user enters too many or
    // too few arguments, or doesn't enter a LIST or MATRIX

    if (argumentNumber > 2)
          ER_throwVar (940);

    if (argumentNumber < 2)
          ER_throwVar (930);

    if (*listPointer != LIST_TAG)
          ER_throwVar (90);

    // get a pointer to the second argument

    itemPointer = next_expression_index (listPointer);

    // decrement listPointer so it points to first element in list

    listPointer--;

    // designate the end of the list of indices that will be
    // pushed to the expression stack

    push_quantum (END_TAG);

    // step through the list, examining each element to see
    // if it matches target item; if it does, add its
    // position to the list

    while (*listPointer != END_TAG)
    {
          // compare_expressions returns 0 if the items match

          if (compare_expressions(listPointer, itemPointer) == 0)
                push_longint (index);

          // advance pointer to the next element

          listPointer = next_expression_index (listPointer);
          index++;
    }

    // list on the top of the stack is in descending order, so
    // reverse the list so items are in ascending order

    push_reversed_tail (top_estack);
    push_quantum (LIST_TAG);

    // estack clean up

    itemPointer = next_expression_index (top_estack);

    while (*itemPointer != END_TAG)
    {
          listPointer = itemPointer;
          itemPointer = next_expression_index (itemPointer);
          delete_expression (listPointer);
    }

} // main
```

*(Credit to Brett Sauerwein for the C version)*

### [3.27] Fast symbolic determinants

The TI-89/92+ may take quite a while to find some symbolic determinants with *det()*. For example, a HW2 TI-92+ takes about 11 seconds to return the determinant of this matrix:

$$\begin{bmatrix} a & b & c \\ d & \sin(e) & f \\ g & \sqrt{h} & i \end{bmatrix}$$

Timité Hassan has written an excellent replacement for *det()* called *det2()*, which returns the determinant in about a second. You can get this routine at http://www.ti-cas.org. You will see similar execution time improvements when the matrix elements are exact numeric expressions involving square roots. Some floating-point determinants may be calculated more accurately.

I made some minor changes to Timité's code; I call this new function *det2a()*. I changed the local variable names, which reduced the code size considerably. I also defined the function *is_0()* within the function, so it is self-contained. I changed the returned error message strings. My modified routine is shown below.

```
det2a(m)
Func
©(mat) symbolic determinant
©T. Hassan
©Modified 4novØ1/dburkett@infinet.com
Local n,k,i,j,q,p,r,is_Ø

Define is_Ø(z)=Func                  © Define local function
 when(z=Ø,true,false,false)
EndFunc

If getType(m)≠"MAT"                  © Test input argument type
 Return  "det2a:Invalid argument"

dim(m)→n                             © Get matrix dimensions
If n[1]≠n[2]                         © Test for square matrix
 Return "det2a:Not square matrix"

n[1]→n                               © Get number of matrix rows
If n=1 Then                          © 1x1 matrix
Return m[1,1]
ElseIf n=2 Then                      © 2x2 matrix
 Return m[1,1]*m[2,2]-m[2,1]*m[1,2]
ElseIf n=3 Then                      © 3x3 matrix
 Return
m[1,1]*(m[2,2]*m[3,3]-m[2,3]*m[3,2])-m[1,2]*(m[2,1]*m[3,3]-m[2,3]*m[3,1])+m[1,3]*(m[2,1]*
m[3,2]-m[2,2]*m[3,1])
EndIf

1→q                                  © N x N matrix, N>3
For i,1,n-3
 m[i,i]→p
 If is_Ø(p) Then
  For j,i+1,n
   If not is_Ø(m[j,i]) Then
    rowSwap(m,i,j)→m
    Exit
   EndIf
  EndFor

  If j>n
   Return  Ø
 EndIf
 For j,i+1,n
  m[j,i]→r
  If not is_0(r) Then
   r*m[i]-p*m[j]→m[j]
   q*⁻p→q
  EndIf
 EndFor
EndFor

Return
1/q*Π(m[k,k],k,1,n-3)*(m[n-2,n-2]*(m[n-1,n-1]*m[n,n]-m[n-1,n]*m[n,n-1])-m[n-2,n-1]*(m[n-1
,n-2]*m[n,n]-m[n-1,n]*m[n,n-2])+m[n-2,n]*(m[n-1,n-2]*m[n,n-1]-m[n-1,n-1]*m[n,n-2]))

EndFunc
```

### [3.28] Fill a list or matrix with a constant

The built-in *Fill* command will fill a list or matrix with a constant expression, but it is not a function and cannot be used in TI Basic functions, nor return the result to the entry line. *fill_lm()* shown below is a true function replacement for *Fill*, and also shows a method to use a single function to return either a list or matrix.

```
fill_lm(e,d)
Func
©(expr,{dim}) or (expr,[row,col])
©Fill matrix or list
©23jun02/dburkett@infinet.com

local τ

gettype (d)→τ

© (This expression is all one line)
when(τ="LIST",seq(e,k,1,d[1]),when(τ="MAT",list▸mat(seq(e,k,1,d[1,1]*d[1,2]),d[1,2]),"fil
l_lm err"))

EndFunc
```

*fill_lm()* returns a filled list or matrix, as determined by the type of argument *d*. If *d* is a list, a list is returned, and if *d* is a matrix, a matrix is returned. The contents of *d* specify the list or matrix dimensions. For example,

| | | |
|---|---|---|
| `fill_lm(a,{3})` | returns | {a,a,a} |
| `fill_lm(a,[2,3])` | returns | $\begin{bmatrix} a & a & a \\ a & a & a \end{bmatrix}$ |

If *d* is neither a list or a matrix, the error string "*fill_lm err*" is returned.

### [3.29] Convert data variables to matrices

You can use the built-in *NewData* command to convert matrices to data variables, but there is no complimentary command to convert data variables to matrices. This function does it:

```
datamat(mσ)
Func
©("name") convert data variable to matrix
©22dec01/dburkett@infinet.com
```

```
local mθ,k,l,d,dl

list▸mat(#mσ[1],1)→mθ                 © Convert first data column to matrix
rowdim(mθ)→d                          © Find row dimension

2→k                                   © Loop through remaining data columns
loop
 #mσ[k]→l                             © Convert column to a list
 dim(l)→dl                            © Find row dimension
 if dl=Ø then                         © Done when row dimension is zero
  exit
 elseif dl≠d then                     © Error if row lengths not equal;
  return "datamat() err"              ©  return error string
 else                                 © Augment current column to matrix
  augment(mθ,list▸mat(l,1))→mθ
 endif
 k+1→k                                © Process next column
endloop

mθ                                    © Return matrix

EndFunc
```

This conversion is useful if you want to use functions on data variables, because data variables cannot be passed as function arguments. Use *datamat()* to convert the data variable to a matrix, then execute the desired function.

To convert a data variable *mydata* to a matrix, use

```
datamat("mydata")
```

Note that the name of the data variable is passed as a string.

The code is straightforward, but is somewhat complicated because we cannot find the number of columns in a data variable. However, I use the fact that a data variable column can be extracted to a list with

```
datavar[k]→list
```

and the dimension of *list* is zero if column *k* does not exist. This condition exits the loop.

The only error checking ensures that all the data variable columns have the same number of rows, since this must be true for matrices. If the row dimensions are not equal, *datamat()* returns the string *"datamat() err"* instead of the matrix. The calling program or function can use *getType()* on the result to determine if an error occurred.

### [3.30] Singular value decomposition (SVD) of a matrix

Singular value decomposition is a powerful tool for a variety of matrix-related tasks, including working with matrices and sets of linear equations which are nearly singular. You can use SVD to remove the singularities and get numerical results, but they may not be the results you expect. In addition, SVD can be used for linear least-squares regression problems. A future version of this tip may demonstrate some of those examples, but for now I will only present a good implementation of SVD, written by Hernan Rivera. For now, if you would like to see how SVD is used, start with chapter 2.6 of *Numerical Recipes in Fortran*, by Press, Teukolsky, Vetterling, and Flannery (http://www.nr.com).

SVD decomposes a matrix *m* with dimensions *a* x *b* (with *a* < *b*) into three matrices *u*, *d* and *v* such that

$$m = u \cdot d \cdot v^{T}$$

*u* is a column-orthogonal matrix with dimensions *a* x *b*. *d* is an *a* x *a* diagonal matrix with positive or zero elements (these are the singular values), and *v* is an *a* x *a* orthogonal matrix. The call for Rivera's *svd()* is

```
svd(m,"u","d","v")
```

where *m* is the matrix to decompose, and "u", "d" and "v" are the names of the variables (as strings) to store the output.

```
svd(aa,á,é,í)
Prgm
© sdv(m,"u","d","v")  m = u*d*vᵀ
© Hernan Rivera

Local st,i,k,n,m,ut,sc

Define sc(st1,n)=Prgm
 Local st2,i
 seq(i,i,1,n)→st2
 SortD st1,st2
 newMat(n,n)→e1
 1→i
 While i≤n
  1→e1[st2[i],i]
  i+1→i
 EndWhile
EndPrgm

eigVl(aa*aaᵀ)→st
eigVc(aa*aaᵀ)→uu
colDim(uu)→n
sc(st,n)
round(uu*e1,11)→uu
eigVl(aaᵀ*aa)→st
eigVc(aaᵀ*aa)→vv
colDim(vv)→m
sc(st,m)
round(vv*e1,11)→vv
min(m,n)→k

1→i
While i≤k
 If (aa*vv)[1,i]*uu[1,i]<Ø Then
  mRow(⁻1,uuᵀ,i)→ut
  utᵀ→uu
 EndIf
 i+1→i
EndWhile

SortD st
newMat(n,m)→dd

1→i
While i≤k
 If st[i]<1ᴇ⁻Ø11
  Ø→st[i]
 √(st[i])→dd[i,i]
 i+1→i
EndWhile

uu→#á
vv→#í
```

```
dd→#é
DelVar e1,uu,vv,dd
EndPrgm
```

Note that *svd()* creates and deletes global variables *e1*, *uu*, *vv* and *dd*; your own variables with those names, if any, will be deleted.

*(Credit to Hernan Rivera)*

### [3.31] Store new list element at index greater than dimension

You can store elements beyond a list's current dimension, if the new index is only one more than the dimension. For example, suppose we have a list of three elements

```
{a,b,c}→mylist
```

then we can store a new fourth element, even though the list has only three elements, like this:

```
d→mylist[4]
```

which results in the list {a,b,c,d}. However, this:

```
f→mylist[6]
```

results in a *Domain error*, since the list has only 4 elements at this point, and we are trying to store a new element to the 6th position.

# 4.0 Graphing and plotting tips

## [4.1] Plot "vertical" lines on graph screen

A vertical line is not a function, and the Y= Editor only plots functions, so you can't really plot a vertical line. There are at least three ways to plot a nearly vertical line as  Y= function.

*Method 1*

Definie a function like this:

```
y1=1E1ØØ∗(x-n)
```

where *n* is the x-coordinate at which to draw the vertical line. To see how this works, consider that we are just plotting the line

y = B(x-n) = Bx - Bn

At x = n, y = 0. Since B is very large, the slope of the line is very large, and the line will appear to be vertical. B must be much larger than the range of interest of *x*.

*Method 2*

A variation on this theme defines B as global variable *_vert*, like this:

```
1ᴇ1ØØ→_vert
```

then you can define various vertical lines in the Y= Editor:

```
y1={function to be plotted}
y2=_vert(x-Ø)
y3=_vert(x-3)
```

This will plot the function, and vertical lines at x = 0 and x = 3

*Method 3*

Use the expression

```
when(x<n,-9ᴇ999,9ᴇ999)
```

to plot a vertical line at x=n. This method may be the best because it eliminates the question of just how big to make the constant 1E100 in methods 1 and 2. For example, to plot a vertical line at x = 7, use

```
y1=when(x<7,-9ᴇ999,9ᴇ999)
```

*(Credit to Kevin Kofler; other credit declined)*

## [4.2] Use *DispG* to force update to Δx and Δy

Δx and Δy are system variables that represent the distance between the centers of two pixels in the graph view window. These variables are a calculated from the system variables *xmin*, *xmax, ymin*, and *ymax*. However the values for  Δx and Δy are not automatically updated when *xmin,* etc., are changed. This is usually not a problem unless you use Δx and Δy in calculations in a program. You can force Δx and Δy to be updated by executing the *DispG* instruction. This effect is shown by this test program:

```
test()
Prgm

clrio

Ø→xmin:5Ø→xmax
Ø→ymin:5Ø→ymax

disp "1. Δx="&string(Δx)
disp "   Δy="&string(Δy)

dispg

disp "2. Δx="&string(Δx)
disp "   Δy="&string(Δy)

1Ø→xmin:2ØØ→xmax
1Ø→ymin:2ØØ→ymax

disp "3. Δx="&string(Δx)
disp "   Δy="&string(Δy)

dispg

disp "4. Δx="&string(Δx)
disp "   Δy="&string(Δy)

EndPrgm
```

This program displays Δx and Δy, both with and without the *DispG* instruction. For the case labeled 1, the values shown are whatever they happened to be before *test()* was run; the assignment of 0 and 50 to the min & max variables has no effect.  For the case labelled 2, the correct values are shown because *DispG* was executed. The min & max values are changed between cases 2 and 3, but the displayed values don't change, because *DispG* isn't used. The final display case 4 shows that Δx and Δy are finally updated, since *DispG* is again used.

So, if you want to use Δx and Δy in calculations in your program, execute *DispG* before you use them, but *after* values are assigned to *xmin*, *xmax*, *ymin* and *ymax*.

## [4.3] Truth plots

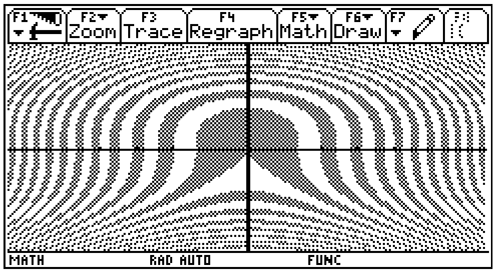
A truth plot is a graphic plot of a function such that a display pixel is turned on when the function is true, and turned off when the function is false. This type of plot is built in to the HP48/49 series, but not the TI89/92+.

This is Andrew Cacovean's version of a program to make a truth plot for an expression:

```
truth(exp1)
Prgm
```

```
Local tlist,xlist,exp2
ClrDraw
DispG
DelVar τy1,τy2
exp1|y=τy2→exp2
exp1|y=τy1→exp1

augment(seq(expr("when("&string(exp1|x=xx)&",τy1,Ø)"),xx,xmin,xmax,Δx*2),seq(exp
r("when("&string(exp2|x=xx)&",τy2,Ø)"),xx,xmin+Δx,xmax,Δx*2))→tlist

augment(seq(x,x,xmin,xmax,2*Δx),seq(x,x,xmin+Δx,xmax,2*Δx))→xlist
For τy1,ymin,ymax,Δy*2
 τy1+Δy→τy2
 PtOn xlist,tlist
EndFor
DelVar τy1,τy2
EndPrgm
```

*exp1* is the expression to be plotted, which must evaluate to true or false. The window variables *xmin, xmax, ymin* and *ymax* must be set before this program is called. This program will work on both the 89 and the 92+, since the program plots to view window coordinates, not absolute pixel coordinates. This program has a hard-coded plot resolution of 2, which means that the function is evaluated at every other y- and x-coordinate. This results in a plot that looks like this, for the expression mod(x^2 + y^3, 4) < 2 for x from -6.5 to 6.5, and for y from -3.1 to 3.2,



Press ON while the program is running to stop it. When the program finishes, the plot is shown until you press [HOME].

Here is a minor variation of the program that sets the window limits as arguments, and also lets you set the plot resolution.

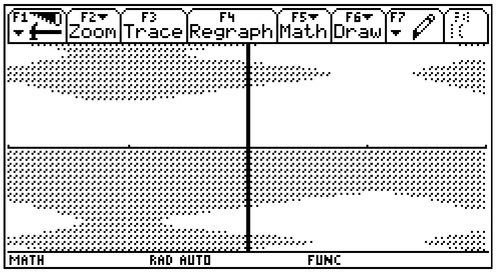
```
truthd(exp1,xxmin,xxmax,yymin,yymax,res)
Prgm
©Truth plot
©Minor change to Andrew Cacovean's truth() program
©12janØØ/dburkett@infinet.com

Local tlist,xlist,exp2

xxmin→xmin
xxmax→xmax
yymin→ymin
yymax→ymax


ClrDraw
```

```
DispG
DelVar τy1,τy2
exp1|y=τy2→exp2
exp1|y=τy1→exp1
augment(seq(expr("when("&string(exp1|x=xx)&",τy1,Ø)"),xx,xmin,xmax,Δx*res),seq(e
xpr("when("&string(exp2|x=xx)&",τy2,Ø)"),xx,xmin+Δx,xmax,Δx*res))→tlist
augment(seq(x,x,xmin,xmax,res*Δx),seq(x,x,xmin+Δx,xmax,res*Δx))→xlist
For τy1,ymin,ymax,Δy*res
 τy1+Δy→τy2
 PtOn xlist,tlist
EndFor
DelVar τy1,τy2
EndPrgm
```

This varies from Andrew's original program only in that the window corner coordinates are passed as arguments, and the plot resolution can be set as a function argument as well. Specifically:

| | |
|---|---|
| exp1: | Expression to be plotted |
| xxmin, xxmax: | Minimum and maximum x-coordinates |
| yymin, yymax: | Minimum and maximum y-coordinates |
| res: | Resolution for both x- and y-axes. res = 1 plots every display point, res = 2 plots every other point, etc. |

Either version of the truth plot program can be very slow, especially when every pixel is tested using *res = 1* in *truthd().* The 92+ LCD has 24,617 pixels, and the 89 display has 12,243 pixels. *truth()* is slow because the function has to be evaluated for each pixel. Setting *res = 2* cuts the time in half, and larger values of *res* reduce the time even more. This plot is for the function sin($x^2$)/x + cos($y^3$)/y < 0, for x and y from -2 to 2, with res = 3. This plot finishes in a few minutes.



*(Credit to Andrew Cacovean)*

## [4.4] Plot data and functions simultaneously

I frequently need to plot both data and a function on the same graph. This is useful when curve fitting and testing interpolation methods. To do this within a program is not difficult, but the method is not immediately obvious. The program below, *plotdemo(),* shows one way to do it.

```
plotdemo(xyspec,xyn,fplot,fpn)
prgm

©Demo program to plot xy-data & 1 function
©24dec99 dburkett@infinet.com
©xyspec: data points matrix or name
```

```
©xyn: xy data plot number
©fplot: function to plot; expression
©fpn: function plot number

local umode,gmode

©Save user's modes
getmode("ALL")→umode
stogdb gmode

©Set graph modes as needed
setmode({"Graph","Function","Split Screen","Full"})

©Clear previous graph contents
clrgraph
clrdraw
fnoff
plotsoff

©Get number of data points
rowdim(xyspec)→xyrows

©Convert xy data points to lists
mat▸list(submat(xyspec,1,1,xyrows,1))→xlist
mat▸list(submat(xyspec,1,2,xyrows,2))→ylist

©Plot the data and zoom to fit
newplot xyn,1,xlist,ylist
zoomdata

©Plot the function & trace
expr(string(fplot)&"→y"&string(exact(fpn))&"(x)")
trace

©Restore user's modes
delvar xlist,ylist
setmode(umode)
rclgdb gmode
disphome

endprgm
```

In this program, *xyspec* is the matrix (or name of the matrix) of the data points to plot. *fplot* is the function to plot, which must have an argument of *x*. *xyn* and *fpn* specify which plot number to use for the data and function plots, respectively. For example, this call:

```
plotdemo(xydata,1,sin(x),2)
```

plots the points in *xydata* as plot number 1, and the the function sin(x) as graph function y2(x).

Beyond all the house-keeping (saving the user's modes, and so on), the critical features of the program are these:

- Convert the matrix data to lists, to use the *newplot()* function. Note that these lists must be global variables to use *newplot(),* so the variables are deleted before the program exits.
- Use *zoomdata after* plotting the data, but *before* plotting the function, to scale the graph to the data.
- Use *expr()* on a string that assigns the input function to a y-function for plotting.
- Plot the function *after* plotting the data.

- Display the graph screen using *trace* so that the data points and function can be traced.

Control returns to the program after the user is done viewing the graph. Both the data points and the function can be traced, as usual, with the [UP], [DOWN], [LEFT] and [RIGHT] cursor movement keys. The user must press [ENTER] or [ESC] to continue running the program. You might consider displaying a dialog box with this information before showing the graph.

Since *newplot()* can only use global variables, *xlist* and *ylist* are deleted before the program exits.

The function is plotted by building a string, for example

```
"sin(x)→y1(x)"
```

then this string is evaluated with *expr().* The number of the y-function is passed as the *fpn* parameter, so if *fpn* = 3, then the string is actually

```
"sin(x)→y3(x)"
```

The function

```
string(exact(fpn))
```

is used to convert *fpn* to a string which is a simple integer with no decimal points or exponential notation, regardless of the current display mode settings.

Unfortunately, this program leaves the plot definition behind for *xlist* and *ylist*. Since the variables are deleted, the plot will not display, but an error will occur the next time the graph window is shown, unless the plot definition is manually deleted. There is no known method to delete the plot definition within the program.

*(Credit to Olivier Miclo)*

### [4.5] 3D parametric line and surface plots

The 89/92+ do not have built-in functions to plot three dimensional line and surface plots. However, S.L. Hollis comes to the rescue, with his routines *view3d()* and *parasurf().* You can get them from his website at *http://www.math.armstrong.edu/ti92/.*

### [4.6] Graphing piece-wise defined functions with "|" operator

This method uses the Y= Editor to define several functions, each of which defines the function over a given range:

y1 = f1(x) | range1
y2 = f2(x) | range2
y3 = f3(x) | range3
etc...

f1, f2 and f3 are the functions for each range. range1, range2 and range 3 are the conditional expressions that define the x-range for each function. '|' is the "where" operator. For example:

y1 = x | x<3
y2 = -(x+3) | x3 and x<5
y3 = 1/2*x | x≥5

Note that this method does not define a single, piece-wise continuous function. Instead, it defines several functions that, when plotted simultaneously, give the appearance of a single function.

*(Credit to Fabrizio)*

### [4.7] Graphing piece-wise defined function with a unit step function

This is another method to plot piece-wise continuous functions. The advantage to this method is that the piece-wise function is defined as a single function, and can easily be integrated, differentiated, or combined with other functions.

First, define the unit step function u(t) as

```
u(t)
func
when(t<0,0,1)
endfunc
```

Electrical engineers know this as the unit step function; mathematicians call it Heaviside's step function.

You can use the unit step function to 'turn on' various graph segments. This is done by multiplying the function by the appropriate unit step function:

1. For x > x1, use u(x - x1)
2. For x1 ≤ x ≤ x2, use u(x - x1) - u(x - x2)
3. For x < x1, use u(x1 - x)

For example, suppose we have three functions f1(x), f2(x) and f3(x). f1(x) is used for x < -1, f2(x) is used for -1 < x < 2, and f3(x) is used for x > 2. Use the unit step function to define the complete function like this:

```
y1(x) = f1(x)*u(-1 - x) + f2(x)*[u(x - (-1)) - u(x - 2)] + f3(x)*u(x - 2)
```

*(Credit to TipDS)*

### [4.8] Plot functions that return lists

You can define functions in the Y= Editor that return lists, and the elements of those lists will be graphed as separate plots. For example, suppose function *f1()* is defined as

```
f1(x)
Func
return {x^2,x^3}
EndFunc
```

then, in the Y= Editor define

```
y1=f1(x)
```

When the graph screen is displayed, both x^2 and x^3 will be plotted as separate plots.

### [4.9] Faster plots of slow functions

The built-in 89/92+ function plotter can be very slow to plot complicated or time-consuming user-functions when the ZoomFit menu item is used. It seems that the plotter evaluates the function twice at each plot point, perhaps to determine the window limit system variables. Plotting time can be substantially reduced by graphing the function as a data plot instead of a function graph. This program does the plotting:

```
plotxy(fname,xl,xh,pn,res)
Prgm
©("f(x)",xlow,xhigh,plot#,xres) plot function using data plot
©leaves global variables xx and yy
©1jul00/dburkett@infinet.com

local dxx,k

©Find x-step and make x-, y-lists
©Change all 239 to 159 for TI89
res*((xh-xl)/239)→dxx
seq((k-1)*dxx+xl,k,1,ceiling(239/res))→xx
seq(expr(fname)|x=xx[k],k,1,ceiling(239/res))→yy

©Set graph window limits
xl→xmin
xh→xmax
min(yy)→ymin
max(yy)→ymax

©Create plot; display graph
fnoff
plotsoff
newplot pn,2,xx,yy,,,,5
dispg

EndPrgm
```

The function arguments are:

*fname* Name of function to be plotted, as a string, with an independent variable of *x*
*xl* Lower limit for independent variable *x*
*xh* Upper limit for independent variable *x*
*pn* Number of data plot to use (1 to 9)
*res* Plot resolution. Set to 1 to plot every pixel, 2 for every other pixel, etc.

The call to plot the function si(x), from 0 to 12, at every pixel, using plot number 2, is

```
plotxy("si(x)",0,12,2,1)
```

After executing this call, the graph is displayed. You may perform all the usual operations on data graphs.

This program has these limitations: You can only plot a single function. The program will over-write current data plot definitions; use the *pn* parameter to specify which data plot to use to avoid this. For simple functions, the built-in function plotter is faster. You must pass the function name as a string, and the independent variable must be *x*. Since this program writes to the graph screen, it must be a program, not a function. The arguments to the *newplot* command must be global variables, not local variables, so these will still exist in the folder from which *plotxy()* is run. You must delete them manually. Since *plotxy()* makes a data plot, not a function plot, you cannot perform Math menu operations such as Zero, Minimum, Maximum and so on.

For even faster plotting, set *res* to 2 or 3 or even larger.

These timing results (92+, HW2, AMS 2.04) show the advantages of using *plotxy().* As an example I use a user-function called si(x), which calculates the sine integral. I plotted over a range of x=0 to x=12, with a resolution of 1.

| | |
|---|---|
| Built-in function plotter, using ZoomFit: | 226 seconds |
| *plotxy():* | 144 seconds (saves 82 seconds or 36%) |
| Built-in function plotter, using manual limits: | 114 seconds |

The fastest method is to use the built-in plotter, and set the *ymin* and *ymax* limits manually.

### [4.10] Faster plots of integral functions

Suppose you want to plot an integral function with an independent variable *x*. Such a function might be in the form

$$\int_{x_0}^{x} f(t)dt$$

where $x_0$ is a constant. For example, the sine integral function is defined as

$$Si(x) = \int_{0}^{x} \frac{\sin(t)}{t} dt$$

You should always first try to find a symbolic expression for the integral; in this case, there is no closed form solution. The most obvious way to plot the function is to define it as a function in the Y= Editor, for example,

```
y1=nInt(sin(t)/t,t,Ø,x)
```

This certainly works, but it is predictably slow, because the integral must be evaluated for each plotted point. For a HW2 92+ with AMS 2.05, it takes about 15 minutes to plot the function with ZoomFit, over a range of 0 to 15 with a plot resolution of one pixel. This is about 3.78 seconds/point. For a HW1 TI89, the time is about 5.41 seconds/point. The efficiency is even worse if we want to plot the function over a range that does not include $x_0$. In other words, we want to plot the function over some range $x_l$ to $x_h$, and $x_l > x_0$. Then, the integral from $x_0$ to $x_l$ is recalculated for every plotted point, but is really the same for each point.

We can considerably reduce the plotting time by recognizing that the integral at each point is just the sum of the integral over the previous plotted points, plus the integral over the interval of the current point. The plotting time is also shortened by generating a lists of x-coordinates and corresponding function values, then plotting with a data plot instead of a function plot. This program, *plotintg(),* implements these ideas:

```
plotintg(fname,xØ,xl,xh,pn,res)
Prgm
©(f(x) or "f(x)",lower ∫ limit,xlow,xhigh,plot#,xres) plot ∫(f(x))
©leaves global variables xx and yy
©11decØØ/dburkett@infinet.com

local dxx,k,p
© dxx is the x-distance between the plotted points
© k is a counter in the loops
© p is (screen width - 2)

©Delete any existing xx and yy variables
delvar xx,yy

©Find x-step and make x-list
getconfg()[dim(getconfg())-1Ø]-2→p     ©Get 'p' for 89 or 92+
res*((xh-xl)/p)→dxx                    ©dxx set by screen width & resolution
seq(k*dxx+xl,k,Ø,floor(p/res))→xx      ©Use floor() so we never try to plot
                                       ©beyond xh

©Find integral at 1st point
©Note that the calculation method depends on whether fname was passed as an
©expression or a string
if gettype(fname)="STR" then
  when(xØ=xl,Ø,nint(expr(fname),x,xØ,xl))→yy[1]
else
  when(xØ=xl,Ø,nint(fname,x,xØ,xl))→yy[1]
endif

©Find integral at remaining points
© For each plotted point, find the integral between the current value of x and
© the previous value of x, then add the previous integral to find the total

©Handle case where fname is passed as a string
if gettype(fname)="STR" then
  for k,2,dim(xx)
    nint(expr(fname),x,xx[k-1],xx[k])+yy[k-1]→yy[k]
  endfor

©Handle case where fname is passed as an expression
else
  for k,2,dim(xx)
    nint(fname,x,xx[k-1],xx[k])+yy[k-1]→yy[k]
  endfor
endif

©Set graph window limits
xl→xmin
xh→xmax
min(yy)→ymin
max(yy)→ymax

©Create plot; display graph
fnoff                     ©Turn off all function plots
plotsoff                  ©Turn off all data plots
newplot pn,2,xx,yy,,,,5   ©Generate xyline plot with dot symbols
dispg                     ©Display the plot

EndPrgm
```

You should run *plotintg()* in Approx mode, and the Graph Mode must be set to FUNCTION. This same program works with either the 89 or the 92+. If you have a global variable called *x*, delete it before running *plotintg()*.

These are the program arguments:

| | |
|---|---|
| fname | The name of the function to be plotted, as an expression or string. The plot is much faster if *fname* is an expression, but some functions can't be passed as expressions.<br>The independent variable must be *x*<br>May also be an expression in *x* |
| x0 | The lower integration limit |
| xl | The minimum value of *x* to plot<br>You must set xl < xh |
| xh | The maximum value of *x* to plot<br>You must set xh > xl |
| pn | The number of the data plot to use. If you have existing data plots in the Y= Editor that you don't want destroyed by the program, choose an unused plot number. |
| res | The plot resolution in pixels, If *res* = 1, then the function is plotted at each pixel. If *res* = 3, then the function is plotted at each third pixel.<br>*res* must be greater than or equal to 1. There is no upper limit, above resolutions of 20 the plot is very coarse. |

Note that you can pass the integrand *fname* as an expression or a string. This flexibility is provided so that expressions plot quickly, but the program still works for integrands that cannot be passed as expressions. This is a limitation of TI BASIC. In general, functions that contain conditional expressions cannot be passed as expressions. If you are not sure that your function can be passed as an expression, just try it - you will quickly get an error message.

For example, to plot a function called *fun1()* from x = 0 to x = 15, use this call:

```
plotintg("fun1(x)",0,15,1,2)
```

which will use plot number 1 and a resolution of 2 pixels. To plot the sine integral shown above, you could use this call:

```
plotintg(sin(x)/x,0,15,1,1)
```

which plots the sine integral from x=0 to x=15, using data plot number 1 and a resolution of 1.

*plotintg()* has these side effects:

- All functions and plots in the Y= Editor are turned off.
- The global variables *xx* and *yy* are created and not deleted. If you have any variables called *xx* or *yy*, they will be overwritten.
- The data plot remains selected in the Y= Editor.
- The plot Window variables are changed and not restored.

The program listing above includes comments which are not included in the actual source file. Remove the comments if you are typing the program instead of using GraphLink.

A design goal for the program was never to attempt to plot beyond the maximum x-value specified by *xh*. This goal is desirable because the function might not even be defined beyond *xh*, or it might be increasing so rapidly that the resulting y-axis scaling would degrade the plot resolution. This is prevented by using the *floor()* function to calculate the number of points plotted. The side effect of this approach is that the function might not be plotted all the way to *xh* if the resolution is not set to 1. This is not catastrophic, in fact, the built-in function and data plotters do the same thing.

The time savings with *plotintg()* can be significant, as shown below. The numbers in parenthesis show the plot time in seconds/point. For the *plotintg()* results, execution times are shown both for passing the integrand as a string (STR) and as an expression (EXP).

| **Integrand function** | **TI89 HW1 built-in plotter** | **TI89 HW1 plotintg()** | **TI92+ HW2 built-in plotter** | **TI92+ HW2 plotintg()** |
|---|---|---|---|---|
| sin(x)/x<br>x0 = xl = 0<br>xh = 15<br>res = 1 | 14:25 (5.41) | STR: 3:59 (1.50)<br>EXP: 3:27 (1.30) | 15:07 (3.78) | STR: 4:21 (1.09)<br>EXP: 2:34 (0.64) |
| ln(x)/sin(x)<br>x0 = xl = 0.1<br>xh = 3.1<br>res = 1 | Not finished in 35 minutes, "Questionable accuracy' shown | STR: 5:14 (1.97)<br>EXP: 3:58 (1.50) | Not finished in 30 minutes, 'Questionable accuracy' shown | STR: 5:38 (1.41)<br>EXP: 3:01 (0.76) |
| 1<br>x0 = xl = 0<br>xh = 1<br>res = 1 | 0:42 (0.26) | STR: 1:27 (0.55)<br>EXP: 1:16 (0.48) | 0:43 (0.18) | STR: 1:42 (0.43)<br>EXP: 1:32 (0.38) |
| x<br>x0 = xl = 0<br>xh = 1<br>res = 1 | 0:47 (0.30) | STR: 1:42 (0.64)<br>EXP: 1:21 (0.51) | 0:73 (0.31) | STR: 1:58 (0.49)<br>EXP: 1:36 (0.40) |

For the first two functions in the table, *plotintg()* is much faster than the built-in plotter. However, for the last two functions, the built-in plotter is faster. Perhaps if the 89/92+ AMS can find a symbolic solution, it uses that solution to evaluate the integral, which would be much faster. Again, always first try to evaluate the integral symbolically - plotting the symbolic solution will always be faster than evaluating the integral.

*plotintg()* does no error checking. Results may be unpredictable if invalid arguments are used.

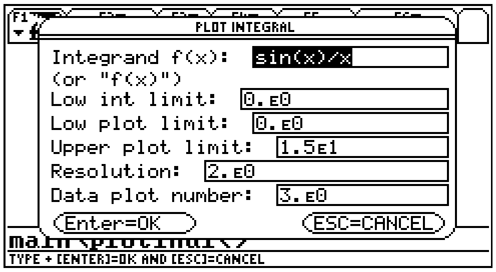
The program shown below, *plotinui()*, provides a user interface to *plotintg()*. It simply prompts for the arguments, does some simple validation of the user input, and calls *plotintg()*.

```
plotinui()
Prgm
©Interface for plotintg() integral plotter
©Calls plotintg() in same folder
©12dec00/dburkett@infinet.com

local umode,et
©umode saves the user's modes, to be restored on exit
©et is the error dialog box title

©Initialize
©Set the error dialog box title
"ERROR"→et
©Save the user's modes
```

```
getmode("Ø")→umode
©Set modes to Function plot, full screen, and Approx
setmode({"1","1","8","1","14","3"})

©Initialize settings if nonexistent
©The settings are saved as global variables, for defaults each time the program
©is executed. If igrand does not exist, the program assumes none of the
©parameters exist, and they are initialized to valid values.
©These global variables are created in the current folder, and are NOT deleted.

if gettype(igrand)="NONE" then
 x→igrand    ©integrand
 Ø→lil       ©lower integration limit
 Ø→lpl       ©lower plot limit
 1→upl       ©upper plot limit
 1→res       ©plot resolution
 3→dpn       ©data plot number
endif

©Main loop
lbl l1

©Convert parameters to strings for dialog box
string(igrand)→igrand
string(lil)→lil
string(lpl)→lpl
string(upl)→upl
string(res)→res
string(dpn)→dpn

©Prompt for all parameters
dialog
 title "PLOT INTEGRAL"
 request "Integrand f(x)",igrand
 text "(or ""f(x)"")"
 request "Low int limit",lil
 request "Low plot limit",lpl
 request "Upper plot limit",upl
 request "Resolution",res
 request "Data plot number",dpn
enddlog

©Convert parameters from strings to expressions
expr(igrand)→igrand
expr(lil)→lil
expr(lpl)→lpl
expr(upl)→upl
expr(res)→res
expr(dpn)→dpn

©Give user the chance to quit here
if ok=Ø:goto l2

©Validate parameters:
©lower plot limit must be less than upper plot limit,
©resolution must be an integer greater than zero,
©data plot number must be an integer from 1 to 99
©If parameter is invalid, display error message and return to input dialog box

©Validate upper & lower plot limits
if lpl≥upl then
 dialog
```

```
  title et
  text "Lower plot limit"
  text "must be less than"
  text "upper plot limit"
 enddlog
 goto l1

©Validate plot resolution
elseif res≤Ø or fpart(res)≠Ø then
 dialog
  title et
  text "Resolution must be an"
  text "integer greater than zero"
 enddlog
 goto l1

©Validate data plot number
elseif dpn≤Ø or dpn>99 or fpart(dpn)≠Ø then
 dialog
  title et
  text "Data plot number must be"
  text "an integer 1 to 99"
 enddlog
 goto l1
endif

©Plot the integral
plotintg(igrand,lil,lpl,upl,dpn,res)
pause
goto l1

©Exit
lbl l2
setmode(umode)     ©Restore modes

EndPrgm
```

This program works on both the 89 and 92+. The input dialog box looks like this on the 92+:



PLOT INTEGRAL

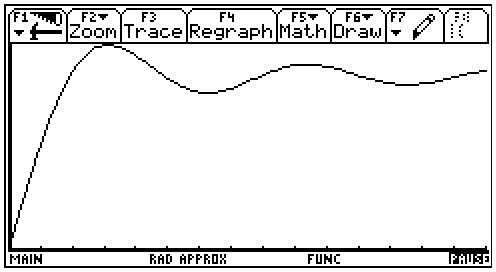
Integrand f(x): sin(x)/x
(or "f(x)")
Low int limit: 0.E0
Low plot limit: 0.E0
Upper plot limit: 1.5E1
Resolution: 2.E0
Data plot number: 3.E0

Note that all the input parameters can be entered, as expected, and that the integrand can be entered as an expression or a string. The program saves the user's modes and restores them when the program is done. Press [ESC] from this dialog box to exit the program, or press [ENTER] to plot the integral.  The plot is shown until you press [ENTER], then the input dialog box is displayed again. This lets you make more than one plot without restarting the program.

The program creates several global variables, so that you do not need to re-enter the parameters to repeat a plot. Therefore, the global variables are *not* deleted. These variables are

```
igrand, lil, lpl, upl, res, dpn
```

Finally, if you're curious as to what the sine integral looks like over the range of zero to 15:



## [4.11] 3D data plots

The 89/92+ can plot 3D functions in a few ways, but they cannot plot 3D data, that is, a set of discrete points in three dimensions. This tip shows how to trick the built-in 3D plotter into drawing a wire frame plot of 3D data points. This is not a true three dimensional plot of the data points, but at least you can get a picture of it. This method will *not* work for arbitrary lists of x-, y- and z-data points: you must have a data point for each xy coordinate.

The trick is to set the plotted z= function as a the matrix of data points, and scale the x- and y-axis maxima and minima so that the x-axis coordinates are the matrix row indices, and the y-axis coordinates are the matrix column indices. The steps to create the plot are:

1. Set the Graph mode to 3D.
2. In the Y= editor, set the desired z= function to *matrix[x,y]*, where *matrix* is the name of the matrix. Push [F1] Format, and choose Wire Frame or Hidden Surface.
3. Push [WINDOW]. Set *xmin* and *ymin* to 1. Set *xmax* to the number of rows in the matrix to plot. Set *ymax* to the number of columns in the matrix to plot. Set *xgrid* to *xmax* - 1. Set *ygrid* to *ymax* - 1. Set *zmin* and *zmax* as needed to span the range of the data in the matrix to plot.
4. Press [GRAPH] to plot the data.

As an example, use this program to create a matrix *mat1* to plot:

```
t()
Prgm
local k,j

for k,1,1Ø
 for j,1,1Ø
  ln(k)*j^2→mat1[k,j]
 endfor
endfor

EndPrgm
```

Following the steps above:

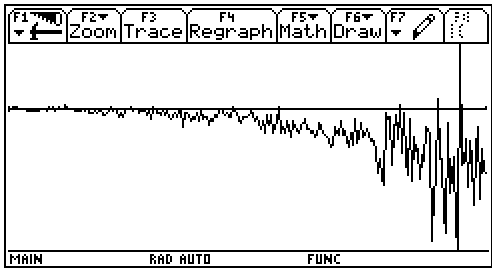
1. Press [MODE] [RIGHT] [5] [ENTER] to select 3D graphing mode.
2. Press [Y=] and move the cursor to the desired z-plot; we'll use z1. Enter *mat1[x,y]* and press [ENTER], so that plot z1 is checked, and the display shows z1 = mat1[x,y].
3. Press [WINDOW]. Move the cursor to *xmin* and enter 1. Move the cursor down to *xmax* and enter 10, since the matrix has 10 rows. Move the cursor down to *xgrid* and enter 9, since xmax - 1 = 9. Move the cursor down to *ymin* and enter 1. Move the cursor down to *ymax* and enter 10, since the matrix has 10 columns. Move the cursor down to *ygrid* and enter 9, since ymax - 1 = 9. Set *zmin* to 0, and set *zmax* to 230.3.
4. Press [GRAPH] to plot the data.

After the data is plotted, you can use the cursor movement keys as usual to change the viewing angle.

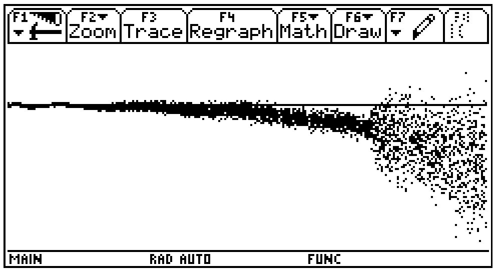
*(Credit to anonymous poster on TI 89/92+ Discussion Group)*

### [4.12] Random scatter plots of 'noisy' functions

The function plot feature of the TI-89/92+ will plot, at most, one point for each x-axis display pixel. If the function is rapidly changing, the resulting graph will not accurately represent the function. For example, the Y= editor was used to make this graph, with a resolution of one:



The graph shows the error for a Chebychev polynomial fit of ln(x) for x=1 to x=2. The y-axis range is about -2.4E-10 to 1.1E-10. However, if we randomly choose many points on the x-axis and plot the function at each point, we get a graph that looks like this:



This type of scatter plot gives us more information about the function. Some oscillation is seen at low values of *x*, then the error slowly increases up to about x = 1.75. Beyond this point, the error suddenly increases. This graph shows at 5000 plotted points.

This program, *scatterf()*, draws this type of scatter plot.

```
scatterf(f,x,xl,xh,yl,yh,n)
Prgm
©("f(var)","var",xmin,xmax,ymin,ymax,its)
©Function scatterplot with random points
©13aug01 dburkett@infinet.com

local k,e

false→e

©Clear graph screen
clrgraph
clrdraw
plotsoff
fnoff
clrio

©Set x- & y-axis ranges
xl→xmin
xh→xmax
yl→ymin
yh→ymax

©Loop to plot the function points
for k,1,n
 if getkey()≠0:exit
 (xh-xl)*rand()+xl→xn
 expr(f&"|"&x&"="&format(xn,"S12"))→yn
 if yn<ymin or yn>ymax then
  true→e
  pxltext "range error",0,0
  disp "Range error: "&string(yn)
 else
  pton xn,yn
 endif
endfor

©Delete global variables
delvar xn,yn

©Display range errors on prgm I/O screen
if e:disp

EndPrgm
```

The input arguments are

| | |
|---|---|
| f | The name of the function to plot, passed as a string. |
| x | The name of the independent variable, passed as a string. |
| xl | The x-axis minimum |
| xh | The x-axis maximum |
| yl | The y-axis minimum |
| yh | The y-axis maximum |
| n | The number of points to plot |

A sample call would be

```
    scatterf("y1(x)","x",1,2,¯3,4,1000)
```

which plots 1000 points of the function *y4(x)* from x=1 to x=2, with a y-axis range of -3 to 4.

*scatterf()* plots each point as it is calculated, so you can see the plot develop. You may stop *scatterf()* by pressing any key. Press [HOME] to return to the home screen display.

*scatterf()* clears the graph screen and turns off all function and data plots. Also, the program I/O screen is cleared. Note that you must specify the y-axis limits. If you set the limits too low, so that not all the points can be plotted, *scatterf()* displays the error message *range error* in the upper left screen corner, but it continues plotting. *scatterf()* also displays the y-value of that caused the range error on the program I/O screen. If range errors occurred, the program I/O screen is automatically displayed when *scatterf()* is finished, or if you press a key to stop *scatterf()*.

There is little point to using *scatterf()* for most functions. *scatterf()* is most useful when the function changes very rapidly at each x-axis pixel coordinate. For best results, thousands of points are typically be plotted.

## [4.13] Bode plots

A Bode plot graphs the magnitude and phase angle of a complex function of frequency. Both the frequency and magnitude are plotted on logarithmic axes, but the phase angle uses a linear axis. Bode plots are commonly used in electrical engineering applications to show the performance of filters or other transfer functions. The 89/92+ do not have built-in logarithmic scaling for function plot axes, but you can easily accomplish this effect with a data plot. In general, the steps are:

1. Define the transfer function G(s), where *s* is the frequency-domain variable.
2. Create a frequency list with logarithmic spacing. You can use the function *loglist()* in tip 3.22 to do this, or you can do it manually.
3. Create a list of the magnitudes of the transfer function at each frequency, by applying the built-in *abs()* function to G(s) evaluated at each frequency.
4. Create a list of the phase angles of the transfer function at each frequency, by applying the built-in *angle()* function to G(s) evaluated at each frequency.
5. Create a list of sequential integers, with as many elements as the frequency list in 2. This list will be the *x* variable in the data plot. This list is easily made with the built-in *seq()* function.
6. For the magnitude plot, define a data plot with the lists in 5 and 3.
7. For the phase plot, define a data plot with the lists in 6 and 3.
8. Display the plots with the ZoomData menu item in the Zoom menu.

Other users have written Bode plot programs that are more sophisticated than this method, and result in better-looking plots, but this method is effective if you only need the occasional plot.

As an example, suppose that we want to plot the transfer function, from 10 to 1000 Hz, of a filter defined by

$$G(s) = \frac{w_0 s}{s^2 + \frac{w_0}{Q} s + w_0^2}$$ where $w_0$ = 100 Hz and Q = 11

This is the general transfer function for a 2nd-order bandpass filter, with unity gain at the center frequency. In general

$$s = jw \quad \text{where} \quad j = \sqrt{-1}$$

(Electrical engineers use *j* for the complex unit, but the TI-89/92+ use *i*).

In the frequency domain, we work with frequencies in radians/second, not Hz, so first we convert the frequencies of interest:

10 Hz = 62.83 rad/sec  
100 Hz = 628.3 rad/sec  
1000 Hz =  6283 rad/sec

Substituting the values in the transfer function we have

$$G(s)\frac{62.83s}{s^2+57.12s+394761}$$

At the calculator command line, we define this function by entering

```
(62.82*s)/(s^2+57.12*s+394761)→g(s)
```

Make sure that Exact/Approx mode is set to Approx, the angle mode is set to Radian, and the Complex Format is set to Rectangular.

Create the logarithmic frequency list using *loglist()* from tip 3.22. We will plot 50 points between the frequencies of 62 and 6300. Since I have *loglist()* stored in the *\util* folder, the command is

```
util\loglist(62,63ØØ,5Ø)→flist
```

and the list is stored in *flist*. *flist* looks like this: {62, 68, 74.6, 81.8, .... 4774, 5237, 6300}

To create the list of transfer function magnitudes at each frequency, use

```
2Ø*log(abs(g(i*w)))|w=flist→mag
```

The expression 20*log() converts the magnitudes to decibels, which are typically used to express transfer function gain. The built-in *abs()* function is used to find the magnitude of the complex result of the transfer function, and *i* is the complex unit, not the letter "i". The magnitudes are stored in the list *mag*.

To create the list of transfer function phase angles, use

```
angle(g(i*w))*(18Ø/π)|w=flist→phase
```

The expression `*(18Ø/π)|` is used to convert the phase angles from radians to degrees. Omit it if you want to plot the phase angle in radians. The resulting list is stored in the variable *phase*.

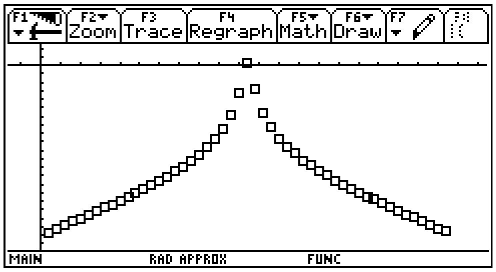
To create the list of sequential integers, use

```
seq(k,k,1,51)→xcoord
```

Note that the number of elements, 51, is one more than the *loglist()* argument of 50, since *loglist()* returns one more element than the number specified.

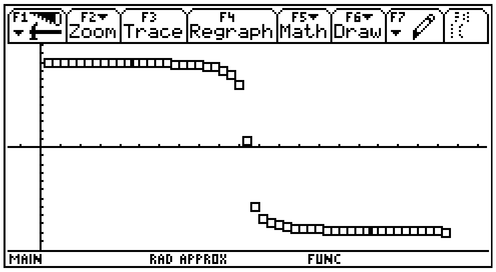
Now that we have the three lists, it is a simple matter to use the built-in data plotting functions to create the plots. Use these keystrokes to set up and display the magnitude plot:

1. Press [Y=] to display the Y= editor
2. Use [UP] and [DOWN] to highlight Plot 1
3. Press [F3] to edit the plot definition.
4. Set the Plot Type to Scatter, set the Mark to Box, set *x* to *xcoord*, and set *y* to *mag*. Push [ENTER], [ENTER] to finish editing.
5. Press [F2] to display the Zoom menu, then press [9] to select ZoomData. This is the resulting magnitude plot on a 92+:



The procedure to display the phase plot is similar:

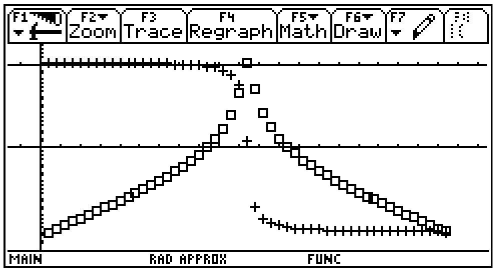
1. Press [Y=] to display the Y= editor
2. Use [UP] and [DOWN] to highlight Plot 2
3. Press [F3] to edit the plot definition.
4. Set the Plot Type to Scatter, set the Mark to Box, set *x* to *xcoord*, and set *y* to *phase*. Push [ENTER], [ENTER] to finish editing.
5. Use [UP] and [DOWN] to highlight the definition for Plot 1, then press [F4] to remove the check mark.
6. Press [F2] to display the Zoom menu, then press [9] to select ZoomData. This is the resulting phase plot on a 92+:



For both the magnitude and phase plots, you can use [F3] to trace the graph and read either the magnitude or the phase at the bottom of the display, as the *yc* variable. However, you cannot directly read the frequency, since the *xc* variable only shows the elements of the *xcoord* list. You can, though, manually display the frequency. Suppose that we want to know the lower frequency at which the magnitude is about -30dB. Push [F3] to trace the plot, then use [LEFT] and [RIGHT] to move the cursor until *yc* is about -30. For this example, *yc* is -29.67 when *xc* is 13. Press [HOME] to return to the Home screen, then enter flist[13] in the entry line. The result is about 188 rad/sec, or 30 Hz.

It is also possible to show the magnitude and phase plots on the same screen, which is useful for determining the relationship between the two results. Use these steps:

1. Create the magnitude plot as described above.
2. Press [F1] to display the Tools menu, then press [2] to select Save Copy As. In the dialog box, set the Type to Picture, and set the Variable name to *magplot*. Press [ENTER], [ENTER] to close the dialog box.
3. Press [Y=]  to display the Y= editor. Select Plot 1 (the magnitude plot) and press [F4] to remove the check mark.
4. Create the phase plot as Plot 2, as described above. You may want to use a different symbol, such as the '+' symbol, so that you can easily distinguish the two plots. Display the phase plot by pressing [F2], then [9] to select ZoomData.
5. With the phase plot still shown, press [F1] to open the Tools menu, then press [1] to select Open. In the dialog box, set the Type to Picture, and select *magplot* as the Variable. Push [ENTER], [ENTER] to close the dialog box. The magnitude plot is shown superimposed on the phase plot, like this:



Here are some more tips on using this method:

- The number of elements in *flist* determines the plot resolution. More elements result in a better-looking plot, but it takes longer to calculate the magnitude and phase. There isn't much point to using more than 240 elements on a 92+, or 160 elements on the 89, since those are the screen widths in pixels. If you use a lot of elements, choose Dot or Box for the symbol type. However, for a quick graph with only a few points, the Box symbol clearly shows each point, even if the points lie on an axis.
- It is sometimes instructive to plot the phase of the transfer function, as a function of magnitude. This is easily accomplished once you have created the *mag* and *phase* lists. Simple create a data plot definition with *mag* as the *x*-variable, and *phase* as the *y*-variable.

*Biographical note*

The Bode plot is named for Hendrik Bode, an engineer and mathematician working at Bell Laboratories in the 1920's and 1930's. Along with Harold Black and Harry Nyquist, Bode was instrumental in developing a comprehensive regenerative feedback theory. He invented techniques for synthesizing networks with desired characteristics, which has applications in telephone line equalization. He also worked with R. B. Blackman and Claude Shannon to model messages and signals in probabilistic terms, which was a new idea at the time.

Hendrik Bode is obviously not the other well-know Bode, the German astronomer Johann Elert Bode, who lived from 1747 to 1826. J.E. Bode compiled an astronomical atlas for 51 consecutive years, and also, unfortunately, his name is associated with a flawed theory to predict the location of planetary orbits.

### [4.14] Completely clear Graph screen

Some programs may require completely clearing the Graph screen of plots, functions, axes and so on. This program does it:

```
graphClr()
Prgm
©Completely clear Graph screen
local ä

StoPic ä
XorPic ä
DelVar ä

EndPrgm
```

The method works by exclusive-ORing the current graph screen with itself, clearing all the pixels. This method is much simpler than using the built-in commands to clear the functions, plots, axes and so on.

*(Credit to Kevin Kofler)*

### [4.15] Delete statistics plots from programs

There is no TI Basic function or command which will delete a statistics plot (the 'plots' in the Y= editor), which is unfortunate, because it is often useful in a program to create plots, then delete them when the program finishes.

The work-around is to save the user's plot definitions with StoGDB before creating the plots, then restore the definitions with RclGDB before the program terminates.

*(Credit to Titmité Hassan)*

# 5.0 GraphLink Cable and Software Tips

### [5.1] Unarchive variables to fix GraphLink transmission errors

This tip applies to GraphLink software for Windows 95/98, beta version, and the gray and black cables. It may also apply to other operating systems and GraphLink software versions.

If you try to send a variable to the calculator, and that variable is locked, you will get a transmission error. Unarchiving the variable allows for successful transmission.

### [5.2] Backing up and restoring individual variables may be safer that Get Backup

It is convenient to use the GraphLink *Get Backup* and *Send Backup* commands to make backups of the calculator memory. However, this may cause problems if some hidden system variables are corrupted, because *Get Backup* copies *all* memory, included the corrupted settings. If you restore with this corrupted backup, the corrupt settings are restored as well.

The symptoms of a corrupt backup may include strange operation, calculator 'lock ups' or 'freeze ups', and the inability to run assembly programs.

Here is a procedure to make a safe, clean backup-up.

1. Use the GraphLink Link, Receive... commands to transfer your calculator variables to your PC. These variables include all the programs and data you want to restore later. Record the folders in which the variables are located. It may help to create directories on your PC with the same names as the calculator folders.
2. Use [MEM], [F1] RESET, All Memory on the calculator to completely reset all memory. This will reset any corrupted system settings.
3. Use the GraphLink Link, Send... commands to transfer the variables from the PC to the calculator. Remember to create the folders you need.
4. Now use the GraphLink Get Backup command to do a complete backup. This backup will not be corrupted

In general, it is not a good idea to make a backup from a calculator which has not been reset in a while.

Note that TI has not confirmed or acknowledged that this is actually a problem, nor that the solution is valid.

*(Credit to Lars Frederiksen)*

### [5.3] Do not restore a HW1 backup to a HW2 calculator

There are two hardware versions of the 89 and 92+ calculators, called Hardware 1 (HW1) and Hardware 2 (HW2). Changes from HW1 to HW2 include a faster processor clock and modified display connections. If you own a HW1 calculator and buy a HW2 calculator, it seems reasonable to use a backup from your HW1 calculator to restore your data and programs to the new HW2 calculator. Unfortunately, there is some evidence that in rare cases this might cause strange operation and lock-ups on the HW2 calculator.

To be completely safe, use the procedure in tip [5.2] to transfer your data and programs from your HW1 calculator to the HW2 calculator. In general this means transferring programs and variables individually with GraphLink, then restoring them individually to the HW2 calculator.

*(Credit to Lars Frederiksen)*

### [5.4] Try Var-Link to fix GraphLink transmission errors

Some ASM programs may crash your calculator, requiring that the batteries be removed. In some cases, GraphLink will not work after this crash. Entering, then exiting the Var-Link screen seems to fix this problem, 100% of the time, according to ZLX. He is using a TI89, AMS 2.05, HW1 and GraphLink 2.0.

*(Credit to ZLX)*

### [5.5] Use TalkTI to develop PC applications which communicate with the 89/92+

*TalkTI* is a set of software components which implement data communcations between a personal computer and TI calculators, or the CBL/CBR. TalkTI only works with Windows operating systems. The TalkTI components can be used with scripting languages such as VBscript or Javascript), or programming languages which are compatible with the Microsoft COM, such as Visual Basic, Visual C++ or Visual J++. The documentation includes examples for these languages. The details of the communications protocol and format are hidden by the software components. TalkTI works with both the gray and black GraphLink cables.

The TalkTI SDK can be downloaded here:

http://education.ti.com/developer/talkti/high/hilight.html

TI hosts a related discussion group:

http://www-s.ti.com/cgi-bin/discuss/sdbmessage.cgi?databasetoopen=calculators&topicarea=
Software+Development+Kit+for+TalkTI&do_2=1

### [5.6] Build your own GraphLink cable (Hess's BCC Serial Link)

A GraphLink cable provides a serial interface between the calculator and a personal computer. This gives you the capability to download programs to the calculator, to back up your calculator's memory on the PC, and to transfer variables between the calculator and the computer. The original TI cable was gray in color, and so was called the gray cable. TI has since released another version, called the 'black cable', which is less expensive, has a higher transfer rate, and seems to be more reliable. The black cable is reasonably priced at $20US, and is readily available, at least in the US.

Even so, there are some good reasons to build your own cable. One reason is education: to learn how such a cable works, and to gain insight into the GraphLink communications protocol. If you live outside the US, TI's cable may not be as readily available, or the cost may be excessive. Finally, if you are an electronic experimenter, you probably already have all the parts you need to build the cable.

Several users have developed alternatives to TI's GraphLink cables. As an example, I have chosen a design by Frank Mori Hess, for several reasons. Frank calls his design the BCC, which is an acronym for 'black-compatible cable', because it is compatible with TI's black cable. The BCC works with TI's GraphLink software; other cable designs require custom third-party software. The BCC is built with inexpensive, non-critical, readily available parts. I have built both BCC versions, and they work quite well. The BCC works only with IBM-compatible personal computers, not with the Apple Macintosh.

You should not attempt to build the BCC unless you have some electronics experience. The BCC is extremely simple, and no users have reported any damage to either the calculator or the PC from a properly constructed BCC. Still, there is always the risk of damage, particularly if the cable is incorrectly assembled. Neither Frank nor I accept any responsiblity for any damage that may result.

Almost all of the information for this tip is taken from Frank's excellent web site:

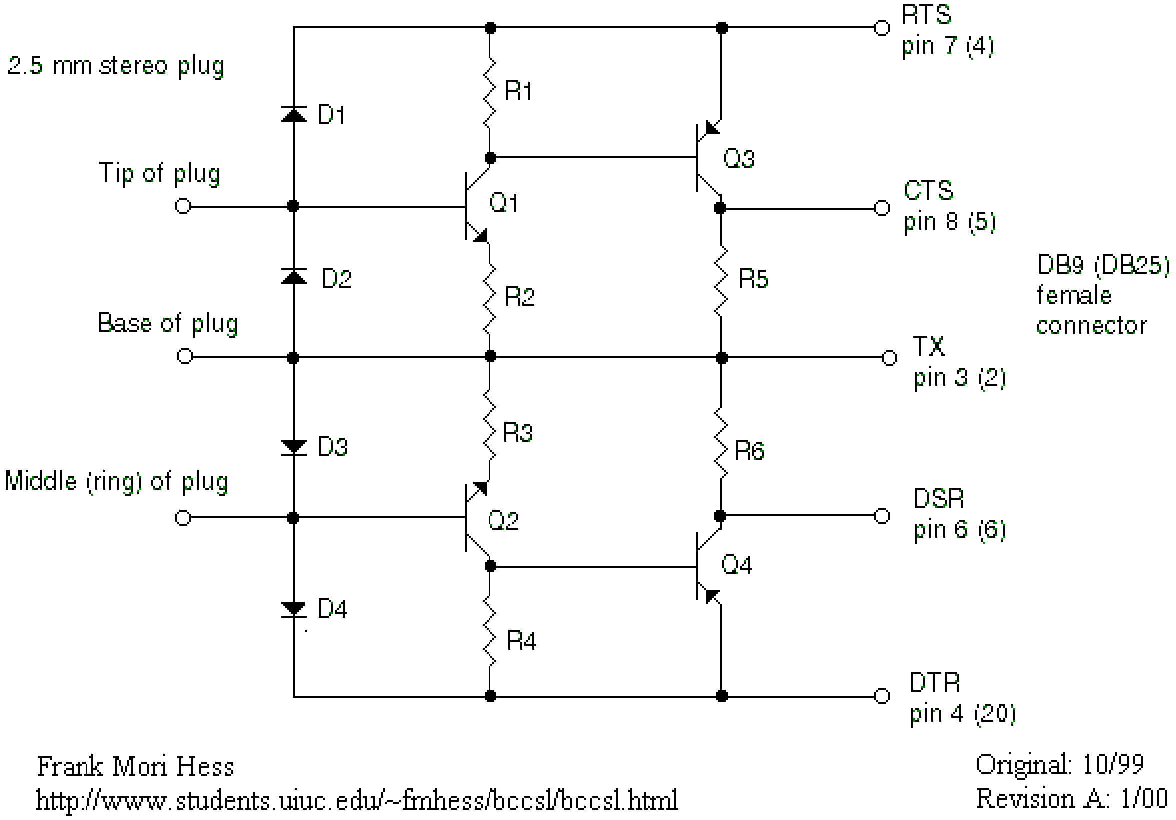
*http://www.students.uiuc.edu/~fmhess/bccs/bccsl.html*

and is reproduced here with his permission. You can email Frank at *fmhess@uiuc.edu*

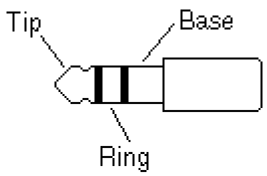
### *The original BCC*

Frank designed two versions of the BCC cable, the 'original' and the 'deluxe'. The original cable works well, but the deluxe cable draws less current and will probably work better with PCs which do not implement true RS-232 signal voltage levels. The deluxe cable adds two transistors to the design, and changes the resistance values. The balance of this section describes the original cable, and the following section describes the deluxe cable.

## "Black Cable" Compatible (BCC) Serial Link for TI calculators



The schematic for the original cable is shown above. The connection to the computer may be either a DB9 or a DB25 connector; the DB25 pin numbers are shown in parentheses. The GraphLink plug which connects to the calculator is a 3-conductor (stereo) 2.5 mm connector. The three conductors are named like this:



The parts list for the original cable:

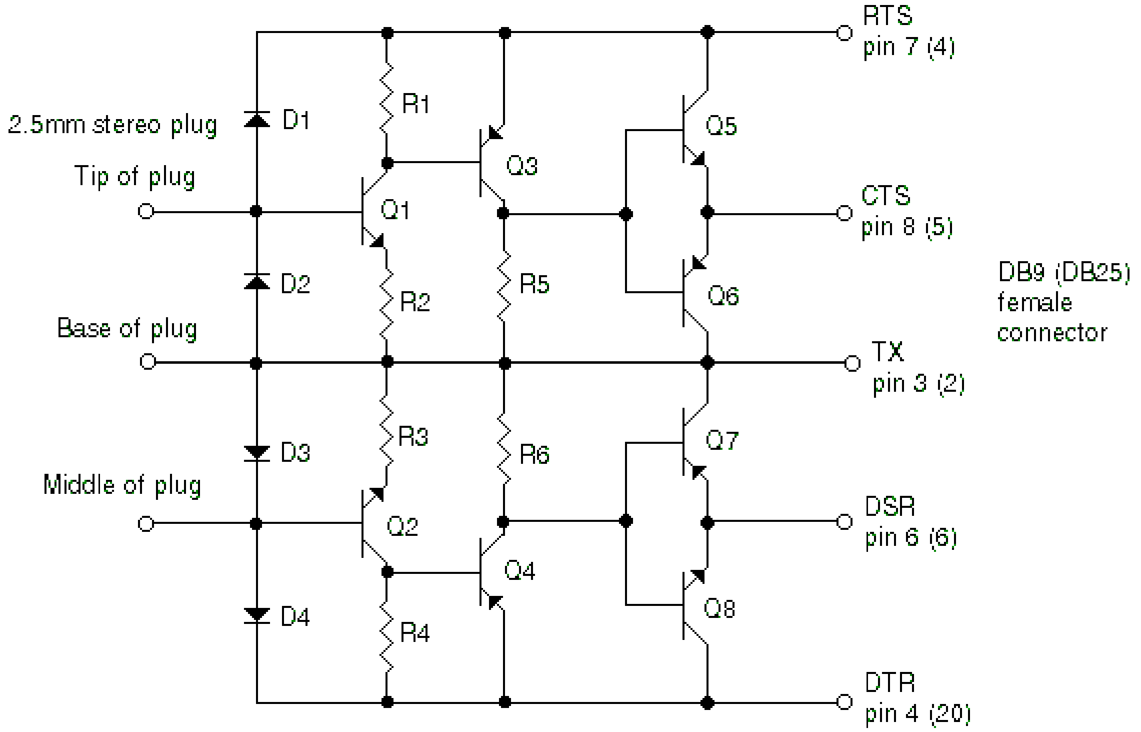
| Designator | Value | Comments |
|---|---|---|
| D1, D2, D3, D4 | 1N914 | May also be 1N4148, 1N4004. Not critical. |
| Q1, Q2 | 2N4401 | May also be 2N3904, or any NPN transistor with at least 25V collector-emitter voltage rating |
| Q3, Q4 | 2N3906 | Any PNP transistor with at least 25V collector-emitter voltage rating |
| R1, R2, R3, R4 | 20kohm, 1/8W, 10% | Any type of resistor will work, also any value from 18K to 24K |
| R5, R6 | 7.5kohm, 1/8W, 10% | Any type of resistor will work, also any value from 5.6K to 9.1K |
| Miscellaneous | Printed-circuit board (PCB)<br>Enclosure<br>2.5mm stereo plug (or jack)<br>DB9 or DB25 female connector<br>Hook-up wire | |

The diodes, resistors, transistors, DB-9 connector and are commonly available from many sources, including Radio Shack, Digikey (www.digikey.com) and Mouser Electronics (www.mouser.com). Mouser carries a very nice 2.5mm jack (161-3307, $1.47) which I use. As an alternative to this jack, you could also make a hard-wired pigtail, connected to the BCC printed circuit board, by cutting off one end of the calculator-to-calculator link cable which comes with the calculator. You can also buy this cable from TI at their on-line store; look under 'Graphing Calculators'.

### *The Deluxe BCC*

The deluxe BCC is quite similar to the original BCC. A push-pull output stage is added and the resistor values are changed, so that the circuit draws less current (meaning longer calculator battery life), and the signal levels at the serial port outputs are increased. The increased level provides better compatibility with some PC serial ports, which do not really comply with the ±12V RS-232 standard.

## BCC–Deluxe Serial Link for TI Calculators



Frank Mori Hess
http://www.students.uiuc.edu/~fmhess/bccsl/bccsl.html

**Original: 5/2001**

The parts list for the deluxe cable:

| Designator | Value | Comments |
|---|---|---|
| D1, D2, D3, D4 | 1N914 | May also be 1N4148, 1N4004. Not critical. |
| Q1, Q2, Q5, Q8 | 2N4401 | May also be 2N3904, or any NPN transistor with at least 25V collector-emitter voltage rating |
| Q3, Q4, Q6, Q7 | 2N3906 | Any PNP transistor with at least 25V collector-emitter voltage rating |
| R1, R2, R3, R4 | 100kohm, 1/8W, 10% | Any type or wattage of resistor will work, also any value from 82K to 120K |
| R5, R6 | 40kohm, 1/8W, 10% | Any type or wattage of resistor will work, also any value from 36K to 43K |
| Miscellaneous | Printed-circuit board (PCB)<br>Enclosure<br>2.5mm stereo plug (or jack)<br>DB9 or DB25 female connector<br>Hook-up wire | |

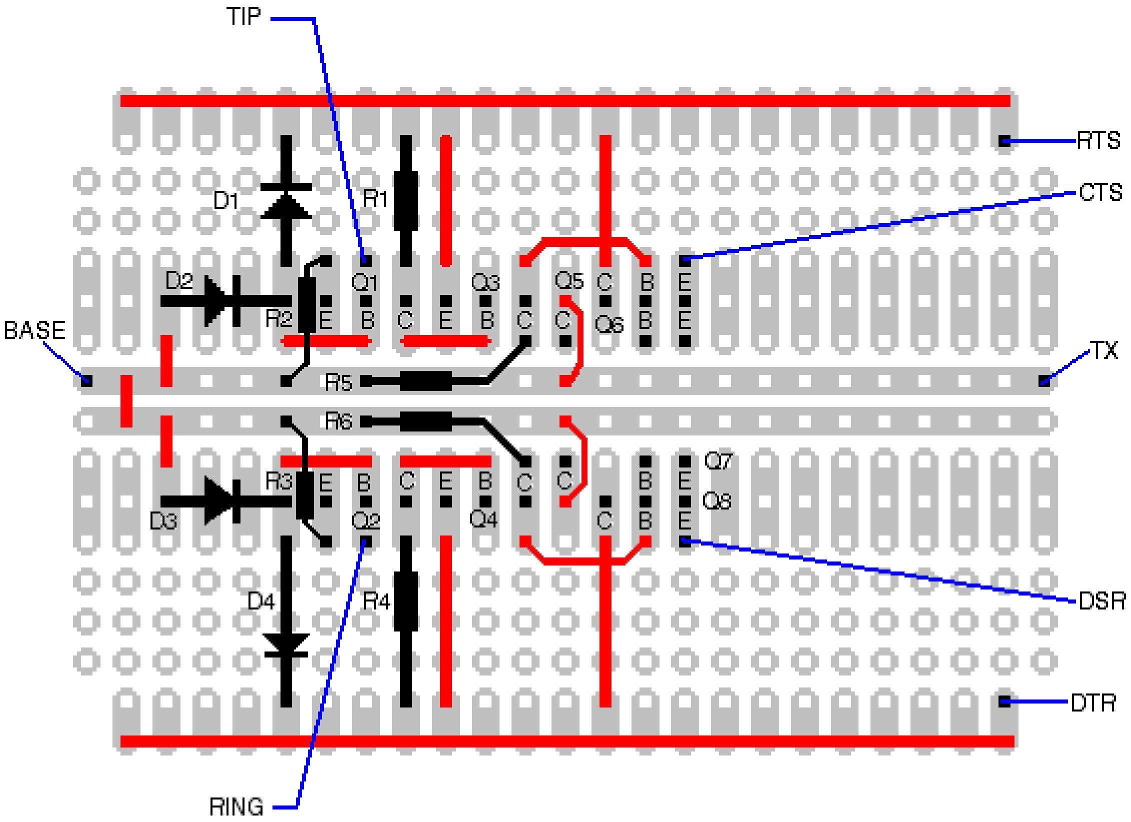
You can build the deluxe cable with standard parts from Radio Shack, as shown in this table:

***RadioShack parts list for BCC deluxe***

| Designator | Value | RadioShack part number | Total cost US$ |
|---|---|---|---|
| D1, D2, D3, D4 | 1N914/1N4148 | 276-1122 (pack of 10) | 1.19 |
| Q1, Q2, Q5, Q8 | 2N4401 | 276-2058 | 1.96 |
| Q3, Q4, Q6, Q7 | PN2907 | 276-2023 | 2.36 |
| R1, R2, R3, R4 | 100kohm, 1/8W, 10% | 276-1347 (pack of 5) | 0.49 |
| R5, R6 | 47kohm, 1/8W, 10% | 276-1342 (pack of 5) | 0.49 |
| Printed-circuit board (PCB) |  | 276-150 | 1.19 |
| DB9 female connector |  | 276-1538C | 1.29 |
| Enclosure | 4" x 2" x 1" | 270-1802 | 2.29 |
| Stand-offs | 10 mm | 276-1381 | 2.19 |
| 4-40 machine screws |  | 64-3011 | 1.49 |
| 4-40 hex nuts |  | 64-3018 | 1.49 |
| 3/32" female jack |  | 274-245 | 1.99 |
| Hook-up wire | 4-conductor 24GA solid phone cable, 1 foot | 278-1310 | 0.15 |
|  |  | Total cost: | 18.57 |

All of these parts should be in stock at any RadioShack, but your particular RadioShack may have to order some of them. Also, prices may vary slightly from store to store. The 4-40 machine screws and nuts are used to mount the DB9 connector to the enclosure. The individual wires of the phone cable are used for jumper wires on the PCB. Considering that the component cost is about $19, and you can buy the TI cable for $20, you aren't saving much money. However, much of the cost is in the enclosure and mounting hardware. You can save some money by using creative alternatives.

This figure shows one possible component layout on the 276-150 PCB:



Components are shown in black, jumper wires in red, and off-board connections are blue. Note that the PCB is not vertically symmetrical; the two rows of single pads are near the top. This layout shows the component side of the board. All components are mounted on this side, but it will be easier to mount some of the jumpers on the reverse side (called the solder side). The two long jumper wires at the top and bottom of the PCB are bare wire, and need only be soldered at the pads where other jumpers or parts are connected. You must use 1/8W resistors for this layout, because 1/4W resistors are probably too big. Note that the collector leads for Q6 and Q7 are separated from the other leads. Also note that the two center traces are connected with a short jumper, so that both are connected to the base and TX. The general assembly procedure is:

1. Lightly polish the copper pads with very fine steel wool or sandpaper, or ScotchBrite. This removes the oxidation and results in better, faster solder joints.
2. Install all jumpers except the Q6 and Q7 collector jumpers. On the component side, install the emitter jumpers for Q3 and Q4, and the base jumpers for Q5 and Q8. Install the remaining jumpers on the solder side of the PCB.
3. Install the diodes.
4. Install Q1, Q2, Q3 and Q4. Pay close attention to the emitter/base/collector lead order, and mount the transistors as close as possible to the PCB, so that they will clear the enclosure sides.
5. Install the resistors.
6. Install Q5, Q6, Q7 and Q8.
7. Install the collector jumpers for Q6 and Q7, on the component side of the PCB.
8. Install the wires for the the off-board connections for the 3/32" jack and the DB9 connector.
9. Assemble the components into the enclosure.

10. Check your work for solder bridges, which are particularly likely with RadioShack PCBs, because the pads and holes are so large.
11. Test the cable.

### *Circuit description*

The GraphLink cable has three conductors. Two of the conductors are data lines, which connect to the tip and the ring of the 2.5mm plug. The third conductor connects to the calculator ground through the base conductor of the plug. Each data line sends and receives data. One data line is used to transmit a '0' bit, and the other transmits a '1' bit. The data lines are asserted low but float high, about +5V above the calculator ground. So, if either the calculator or the PC asserts a data line, both ends of the line are pulled low. Both data lines have a pull-down resistor of about 10Kohm in the calculator, so the transmitting circuit must sink about 0.5mA. For the moment, assume that the two calculators (a 'transmitter' and a 'receiver') are connected with the GraphLink, then these steps are used to transmit a data bit:

1. Initially both data lines are floating high. This is the idle state.
2. The transmitter asserts one of the data lines low, to transmit a bit.
3. The receiver asserts the other data line to indicate it has received the bit.
4. The transmitter stops asserting low.
5. The receiver stops asserting low.
6. Both calculators are back in the idle state, and the next bit may be transmitted.

Note that precise timing is not required, because this is an asynchronous data link.

Since the GraphLink is designed for communications between two calculators, the purpose of the GraphLink cable circuit is to use the computer's serial port to emulate a calculator. This is accomplished with five of the serial port pins. The RX and TX lines do not receive and transmit data, as is usually the case with RS-232 communications. Instead, the TX line is perpetually set to its low voltage state, and sinks current from the cable circuit. The RX line is not used at all. The RTS and DTR lines are used to transmit data from the PC, and the CTS and DSR lines receive data from the calculator. Since the upper and lower halves of each BCC link circuit are identical, we need only describe the upper half.

This table describes the four conditions the BCC circuit must apply.

| Case | Calculator asserting Low? | State of RTS line | Result at data line (Plug tip and CTS) |
|---|---|---|---|
| #1 | No | HIGH (Positive voltage with respect to serial port ground | HIGH (plug tip 5V with respect to calculator ground, CTS positive with respect to serial port ground). |
| #2 | No | LOW (Negative voltage with respect to serial port ground) | LOW (plug tip 0V with respect to calculator ground, CTS negative with respect to serial port ground.) |
| #3 | Yes | HIGH | LOW |
| #4 | Yes | LOW | LOW |

In cases 3 and 4, the calculator is either transmitting a data bit or acknowledging receipt of a bit, so, regardless of the state of the RTS line, the CTS line must be low so that the PC reads the low level. In cases 1 and 2, the calculator is either in the idle state, or the 'other' data line is asserted low, so the plug tip and CTS must follow the RTS line state. The BCC circuit implements these cases as follows.

*Case 1:*
RTS is at about +12V relative to serial port ground.  The plug tip is at about 5V, relative to calculator ground, so Q1's base is at about -7V relative to the serial port ground. This turns Q1 on which in turn turns Q3 on, so the CTS line is only below RTS by Q3's Vce drop, or, essentially, at RTS, so CTS is high.

*Case 2:*
RTS and TX are at the same voltage, so neither Q1 nor Q3 can turn on. D1 pulls the plug tip low, and pull-down resistor R5 pulls CTS low.

*Case 3:*
The plug tip is low, so neither Q1 nor Q3 turn on. R5 pulls CTS low.

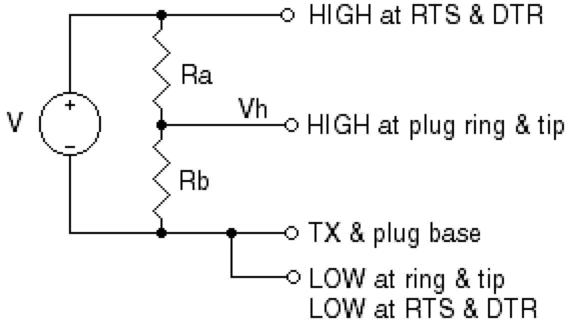
*Case 4:*
Neither Q1 nor Q3 can turn on. R5 pulls CTS low.

The deluxe BCC operates in much the same way, except that Q5 and Q6 form a push-pull output stage to drive the CTS line.

***Testing the cable***

The simplest way to test the BCC is to plug it into you calculator and computer, and try it. Download and install the GraphLink software from the TI website. From the *Link* menu, select the proper serial port and choose the *Windows only (black cable)* option from the *Cable type* submenu. If you can successfully complete the *Get Screen* operation from the *Link* menu, then the cable works. If the *Get Screen* operation fails, immediately unplug the BCC from the calculator and the computer. While both the computer serial port and calculator I/O port are fairly robust, you minimize the chance of damage, to either one, by unplugging everything. If the operation fails, read the rest of this section, and see the *Troubleshooting* section below.

You can also test the BCC before you connect it to the calculator and computer, with this simple test circuit, a voltmeter and some test leads:



The voltage source V may be a power supply or even a 9V battery. The two resistor Ra and Rb set voltage Vh, and simulate the 10K source impedance of the calculator data lines. For a desired source impedance Rs and output voltage Vh, use these equations to find Ra and Rb:

$$Ra = \frac{Rs \cdot V}{Vh} \qquad Rb = \frac{Rs \cdot V}{V - Vh}$$

which are found as the solutions to

$$Rs = \frac{Ra \cdot Rb}{Ra + Rb} \qquad Vh = V \frac{Rb}{Ra + Rb}$$

For example, if we use a 9V battery, and want a 5V output voltage with 10K source impedance, then Rs = 10K, V = 9 and Vh = 5, and we find Ra = 18K and Rb = 22.5K. The nearest standard 5% tolerance values are 18K and 22K, which results in an output voltage of 4.95V and a source impedance of 9.9K, which is close enough. It is simpler and just as effective to use two 20K resistors for Ra and Rb.

The 'tip' and 'ring' sections of the circuit are tested separately, following cases 1 to 4 as described above. The tables below show the correct results, as well as my measured results when using a 9V battery. Note that V+ is the positive terminal of the battery (or power supply), and V- is the negative terminal.

***Test results for 'tip' circuit***

| Case | Calculator asserting low? | State of RTS line | Result at data line (Plug tip and CTS) |
|---|---|---|---|
| #1 | No<br>(Connect tip to Vh) | High<br>(Connect RTS to V+) | tip: HIGH, Vh<br>CTS: HIGH; about 9.3V |
| #2 | No<br>(Connect tip to Vh) | Low<br>(Connect RTS to V-) | tip: LOW, about 0.6V<br>CTS: LOW, about 0V |
| #3 | Yes<br>(Connect tip to Tx) | High<br>(Connect RTS to V+) | tip: LOW, about 0V<br>CTS: LOW, about 0V |
| #4 | Yes<br>(Connect tip to Tx) | Low<br>(Connect RTS to V-) | tip: LOW, about 0V<br>CTS: LOW, about 0V |

***Test results for 'ring' circuit***

| Case | Calculator asserting low? | State of RTS line | Result at data line (Plug tip and CTS) |
|---|---|---|---|
| #1 | No<br>(Connect ring to Vh) | High<br>(Connect DTR to V+) | ring: HIGH; Vh<br>DSR: HIGH; about 9.3V |
| #2 | No<br>(Connect ring to Vh) | Low<br>(Connect DTR to V-) | ring: LOW, about 0.6V<br>DSR: LOW, about 0V |
| #3 | Yes<br>(Connect ring to Tx) | High<br>(Connect DTR to V+) | ring: LOW, about 0V<br>DSR: LOW, about 0V |
| #4 | Yes<br>(Connect ring to Tx) | Low<br>(Connect DTR to V-) | ring: LOW, about 0V<br>DSR: LOW, about 0V |

### *Troubleshooting*

If the BCC cable does not work, first verify that:

1. You are using the latest version of the TI GraphLink software.
2. You have selected the Black Cable option in the GraphLink software.
3. You have selected the correct serial port for your computer.
4. You have reasonably fresh batteries in the calculator.
5. Both the GraphLink and serial port connections are secure. On the TI-89, in particular, the GraphLink cable connector must be *firmly* pushed into the calculator connector.

If all this seems correct, check the BCC circuit itself. Most circuit flaws are caused by mis-wiring. In particular:

1. Verify that the diode polarities are correct.
2. Verify that the transistor connections (base, emitter and collector) are correct.
3. Verify that the plug, tip and base connections are correct.
4. Verify that the serial port connections (RTS, CTS, TX, DSR and DTR) are correct, *for the connector you are using - either the 9- or 25-pin.*
5. Verify that all component connections are correct. Use a voltmeter with a continuity function (beeper). For example, verify that RTS is connected to D1, R1 and Q3's emitter. Verify that the plug base connects to D2, D3, R2, R3, R5, R6 and TX, and so on.
6. Verify that the correct resistor values are in the right places.
7. Check the schematic and verify that you have, in fact, built the circuit shown.

Some circuit flaws are caused by soldering problems:

1. Carefully check the circuit for shorts (unintentional solder bridges between pads or leads).
2. Check all solder connections for cold joints. A good solder joint has a shiny, smooth appearance. A cold joint has a dull, rough appearance. A cold joint may be mechanically solid but may not be a functional electrical connection.

If all this fails to get the BCC working, have someone else check your work. This is helpful even if (particularly if!) the person has no electronics experience. Sometimes, just explaining the circuit to someone else will make a mistake obvious.

It is remotely possible, but unlikely, that failed parts cause the BCC fault. Transistors and diodes can be damaged or destroyed outright by the heat from excessive soldering times. Again, the most likely cause of failure is mis-wiring.

### *Comments*

The serial port input lines (CTS and DSR) have a relatively low impedance of about 5Kohm. The are *not* high-impedance inputs.

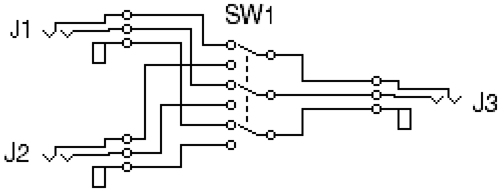
If either of the data lines (plug or ring) are asserted low, calculator operation may become sluggish because the calculator thinks that communications is in progress. It appears that the GraphLink software does not set the DTR and RTS lines high on all computers when the software is started. This may prevent the calculator from being put into 'receive' mode, and it may also prevent sending programs using the SEND button in the GraphLink windows. This can usually be fixed by executing *Recieve* or *Get Screen* from the GraphLink menu.

If you use the GraphLink software to develop CBL or CBR programs, you may want to build the switch-box circuit (described in tip [5.7] below) into your BCC.

*(Credit to Frank Mori Hess for the cable designs and documentation)*

## [5.7] GraphLink switchbox aids CBR/CBL programming

If you use the GraphLink software to write TIBasic programs for the CBR or CBL, you do a lot of cable swapping during program development. You must connect the calculator to the GraphLink cable to download the software, then connect the calculator to the CBL to test your software. This cable swapping wears out the cable and connectors, and can be avoided with a simple switchbox:



J1, J2 and J3 are female 2.5mm stereo jacks, and SW1 is a triple-pole, double-throw (3PDT) toggle switch. SW1 simply switches the three GraphLink cable conductors between J1 and J2, so the calculator connects to J3, the PC connects to J1, and the CBL connects to J2. Even though you must remember to put the switch in the proper position, the switchbox still eliminates the cable swapping.

Because of the low currents involved, SW1 should be a 'dry circuit' switch, with gold contacts. Unfortunately, 3PDT switches with gold contacts are not easy to get. The alternative is to use a switch with silver-plated contacts. Eventually, the silver will oxidize and communications will fail, and the switch will need to be replaced. However, I have been using such a switch for over a year with no failure yet. A typical switch is the C&K 7301SYZQE, which can be obtained from Mouser Electronics (www.mouser.com) as their part number 611-7301-001. Mouser also carries a high-quality 2.5mm stereo jack, their part number 161-3307.

If you build your own GraphLink cable, as shown in tip [5.6], you can build this switchbox into the same enclosure and save a 2.5mm jack.

## [5.8] Opening variables in GraphLink changes some characters

Some characters in strings are changed by the GraphLink software if file variables using these characters are opened. This is a problem if the program is distributed to users as a file created with GraphLink. The characters will not be converted if the program is simply sent from GraphLink with *Link Send*, and the program will work correctly. However, if the user first opens the program file, then the characters are converted and the program will not work. This tip applies to AMS 2.05 and the PC version 2.1 of the GraphLink software.

Characters with these codes are all converted to "?", which is char(63):

1, 3 - 9, 11, 14-17, 19 - 21, 23 - 27, 127, 167, 170, 174, 182, 198, 208, 222

Also:

char(12) is changed to char(126)
char(166) is changed to char(124)
char(184) is changed to char(183)
char(247) is changed to char(47)

And note that:

- char(2) can be copied as a string, but will locate the cursor when pasted.
- char(10) pastes as the line feed symbol in TI Basic programs on the calculator, but pastes as an actual line feed in GraphLink.
- char(13) pastes as the carriage return symbol in TI Basic programs on the calculator, but pastes as an actual carriage return in GraphLink.

For the character codes, refer to the table *TI-89 / TI-92 Plus Character Codes* in Appendix B of the TI-89/TI-92+ *Guidebook*.

You can create a special character by using *char(code)* at the entry line, where *code* is the numeric code for the character. For example, to create the filled-square character:

1. Enter *char(16)* in the entry line and press [ENTER]. The character string is placed in the first answer level of the history display.
2. Press [UP] to select the character string.
3. Press [COPY] to copy the string. [COPY] is [DIAMOND] [C] on the TI-92+, and [DIAMOND] [SHIFT] on the TI-89.
4. The special character string is now in the clipboard, and can be pasted where needed, for example, in a program or function in the program editor.

The solution to the character-conversion problem is to avoid the procedure above, and instead use the *char()* function to create any special characters which are needed in your programs. If you do use the procedure, then you should at least warn the user not to open the program files in GraphLink before sending them to the calculator.

I used this process to determine which characters are changed by GraphLink:

1. Create a string of characters to test at the entry line. For example, *char(3)&char(4)&char(5)* creates a string of three characters. Save the string to a variable called *l1*.
2. Receive *l1* from the calculator from the calculator with GraphLink. Open *l1* in GraphLink and copy the string to the clipboard.
3. Open a new program file in GraphLink called *tstr()*, and paste the string. *tstr()* saves the pasted string to variable *l2*, as shown below.
4. Send *tstr()* to the calculator with GraphLink and run it, to save the string to variable *l2*.
5. Use the *strcompr()* function shown below to compare the original string *l1* to the (possibly) modified string *l2*, and return the differences, if any.

This is the program *tstr()*, which is used to save the converted string to *l2*.

```
tstr()
Prgm
"???"→l2       © Paste string l1 in place of "???"
EndPrgm
```

This is *strcompr()*, which compares the two strings:

```
strcompr(α,β)
Func
©Compare strings α, β; return differences as character codes
©1jan2ØØ2/dburkett@infinet.com

Local k,res,c

if dim(α)≠dim(β):return "dim error"

{}→res

For k,1,dim(α)
 if mid(α,k,1)=mid(β,k,1) then
  {Ø}→c
 else
  {ord(mid(β,k,1))}→c
 endif
 augment(res,c)→res
EndFor

return res

EndFunc
```

*strcompr()* returns a list with one integer for each character in the strings. If the corresponding characters are the same, zero is returned. If the characters differ, the character code for string *l2* is returned. For the example I'm using, this call

```
strcompr(l1,l2)
```

returns {63,63,63}, indicating that none of the converted string characters match the original characters, and they have all been converted to char(63), which is "?".

### [5.9] GraphLink software sends either TI-89 or TI-92 Plus files

There are two versions of TI's GraphLink software, one for the TI-89 and one for the TI-92 Plus. By default, the *Link Send* dialog box shows only the file types for that particular calculator model, for example, GraphLink for the TI-92 Plus shows only file types of .9x*. However, you can send TI-89 files to a TI-92 Plus, and vice versa, by one of these methods with either GraphLink software version:

1. Type the file name directly in the *File Name* text box, if you know the name. Include the file type suffix.
2. In the *List Files of Type* drop-down menu, choose *All Files (*.*)*, then choose the desired file.
3. In the *File Name* text box, type *.89* to list all TI-89 files, or type *.9x* to list all TI-92 Plus files.

Some programs and functions are specific to the particular calculator model, but most functions and programs will run on either model. The file suffix is assigned when the programmer saves the file, so files saved from GraphLink for the TI-89 will have a .89* suffix by default.

Note also that most other GraphLink operations work with either calculator; for example, the screen grab and calculator backup and restore functions work correctly, regardless of which calculator is actually connected.

### [5.10] Run GraphLink under Windows XP

Various visitors to TI's discussion groups have shared their tips on getting the Graph Link and TI Connect software running under Windows XP.

John Gilbertson notes that Microsoft has acknowledged a problem with .vxd files running in Windows XP. During setup of the link software, do not indicate that you are using a black Graph Link cable, as this will install a .vxd file and screw things up. Run the software once using the grey Graph Link cable setting, then you can change it to the black cable and it should work.

Joseph Thornton reports that TI Connect and the Graph Link software work in Windows XP if they are set to run in Windows 98 compatibility mode. To do this, go to the location where the program is installed (usually C:\Program Files\TI Education\TI Connect\) right-click the program executable file (i.e. TIConnect.exe) and click properties, then click the Compatibility tab, then click "Run this program in compatibility mode for Windows 98 / Windows ME", then check the "Disable Visual Themes" box, and finally click OK. Repeat for all the .exe files in the TIConnect folder.

Mr. Thornton also reports that as of this writing (Feb '02), the USB Graph Link cable doesn't work under Windows XP. He suggests using TiLP instead if you have too many problems with TI Connect. Bill Gorder reports success by running the program executable file from a command prompt (DOS) box.

Andrew Farris reports that Windows XP has a problem running 16 bit programs that are too many directories deep. He installed the link software to C:\Program Files\TI\ instead of C:\Program Files\TI Education\TI-Graph Link 89\ and it worked for him.

*(Credit to Ray Kremer)*

### [5.11] Improve font size of printed program source code

Cass Lewart sends this tip to get more readable source code print-outs from TI Connect. It also works with TI GraphLink.

*"Printing a program with TI Connect (File/Print) results in a very small font, which cannot be changed. However, pasting the program into, e.g., Word (From TI Connect File/Open/Select All/Copy) then opening Word and Paste gives many more options. It requires selecting TI92Pluspc or one of the other scalable TI fonts which are installed on the desktop during TI Connect installation. These fonts have a full range of sizes, bold or italic."*

Obviously the same technique works with any word processor or page layout program.

*(Credit to Cass Lewart)*

# 6.0 Math Tips

## [6.1] Simulate *poly* function of TI-85/86

The *poly* function of the TI-85 and TI-86 can be simulated with this function:

```
czeros(polyeval(list,x),x)→poly(list)
```

This defines a function *poly()* which returns the real or complex roots of a polynomial in *x* defined by the coefficients in the argument list. The coefficients are in order of descending degree. For example,

```
poly({1,-11,3Ø})
```

solves for *x* in the polynomial $x^2 - 11x + 30 = 0$, and returns {5, 6}. Remember to include the zeros for powers with zero coefficients. For example, the list would be {4,0,-1,-2} for the polynomial $4x^3 - x - 2$.

*(credit to Ray Kremer)*

## [6.2] Use rectangular complex mode for faster results

Seemingly simple vector calculations can execute very slowly on the 89/92+ in Polar and Exact or Auto modes. This example

$$\frac{(250\angle 85)*(121\angle 3)}{(250\angle 85)+(121\angle 3)}$$

returns the correct result in less than 1 second in Approx, Degree and Polar modes. However, in Exact or Auto modes, the calculator might seem to 'hang up', but really the answer just takes a long time to calculate. A Texas Instruments TIcares representative offers this explanation.

> *Thank you for your recent correspondence with Texas Instruments. Yes this behavior is possible: We internally calculate with expressions in the rectangular forms that are then converted to a polar form. Converting from a very complicated rectangular form to polar form in the exact or auto mode, can be extraneous on our TI-89 operating systems. Internal complex calculations are done in rectangular form. So, the polar complex numbers are converted to rectangular form. Then, the computation (a * b) / (a + b) is performed. Finally, the resulting rectangular complex number is converted to polar form. It's a difficult process. The result takes about 45 minutes on my TI-89 and looks like this.*
>
> *(30250*sqrt(62500*(cos(3))^2*(cos(5))^4+(125000*(sin(5))^2*(cos(3))^2+60500*sin(5)*cos(3)+14641)*(cos(5))^2+60500*sin(3)*cos(5)+62500*(sin(5))^4*(cos(3))^2+60500*(sin(5))^3*cos(3)+14641*(sin(5))^2+62500*(sin(3))^2)/(60500*sin(3)*cos(5)+60500*sin(5)*cos(3)+77141) < atan((121*cos(5)+250*sin(3))/(250*cos(3)*(cos(5))^2+sin(5)*(250*sin(5)*cos(3)+121))))*
>
> *One of the important lessons in computer algebra is that simple looking input may generate very large exact output and may take a great deal of time to compute. If fast times or small output is desired, it is sometimes best to interrupt a computation and approximate it. The approximate answer*

*(103.416 < 27.1821) is generated in about 1 second.*

### [6.3] Improving floating-point solutions to simultaneous equations

Round-off errors in matrix calculations can create errors larger than you might expect when finding solutions to simultaneous equations with *simult().* The closer the matrix is to being singular, the worse the error becomes. In many cases this error can be reduced, as follows.

Suppose we want to find the solution vector *x* in

$$A \cdot x = b \qquad [1]$$

We would use

```
simult(A,b)→x
```

However, because of round-off error, this function really returns *x* with an error of *dx*, which results in *b* being in error by *db*, or

$$A \cdot (x + dx) = b + db \qquad [2]$$

Subtracting [1] from [2], we get

$$A \cdot dx = db \qquad [3]$$

Solve [2] for db, and substitute into [3] to get

$$A \cdot dx = A \cdot (x + dx) - b \qquad [4]$$

Since we know everything on the right-hand side of [4], we can solve [4] for dx:

$$dx = A^{-1} \cdot A \cdot (x + dx) - b \qquad [5]$$

So, if we subtract *dx* from the original solution (x+dx), we get a better estimate of the real solution x.

Usually, equation [5] is evaluated in double precision, with the intent of getting results that are at least accurate to single precision. Even though the 89/92+ do not *have* double precision arithmetic, this process still results in some improvement. It is also common to apply the improvement process several times, to ensure convergence and to find an optimal solution. However, the limited precision of the 89/92+ usually prevent this type of repetitive improvment. Repeating the improvement process results in a solution with more error.

Here is a function that returns the improved solution:

```
simulti(A,b)
func
local x

simult(A,b)→x
x-(A^⁻1*(A*x-b))

Endfunc
```

As an example, consider using *simulti()* to find the coefficients for the Langrangian interpolating polynomial. This is just the polynomial that fits through the given points. A polynomial of degree n-1 can be fit through n points. This function returns the coefficients as a list, given the x-y point coordinates in a matrix:

```
polyfiti(xyd)
func
©Find coefficients of nth-order polynomial, given n+1 xy points
© 18marØØ/dburkett@infinet.com

local a,k,n,xd

rowdim(xyd)→n

seq(k,k,n-1,Ø,⁻1)→a
seq(mat▸list(submat(xyd,1,1,n,1))[k]^a,k,1,n)→a

mat▸list(simulti(a,submat(xyd,1,2,n,2)))

Endfunc
```

This table shows the results of the fit for fitting a function with and without improvement. The function is f(x) = tan(x), *x* in radians. *x* ranges from 0.4 to 1.4, with data points every 0.1 radians. This means that 11 points were fit to a 10th-order polynomial.

| | | |
|---|---|---|
| Without improvement: | RMS residual: | 3.5E-8 |
| | Maximum residual: | 3.5E-6 |
| | Minimum residual: | 3.4E-10 |
| With improvement: | RMS residual: | 7.4E-11 |
| | Maximum residual: | 5.9E-9 |
| | Minimum residual: | -7.8E-10 |

The residual is the difference between the actual b-values and the calculated b-values. The RMS residual is a deviation measurement of all the residuals, and the minimum and maximum residuals are the most extreme residuals for all the fit points.

For this function and data, the improvement results in several orders of magnitude in both the RMS residuals and the extreme residuals. Without the improvement, the calculated results are only accurate to about five significant digits. With the improvement, the results are accurate to at least 8 significant digits.

### [6.4] Gamma, log-gamma and factorial functions

The gamma function is one of many 'special functions' that occur as the result of integrations, limits, sums and products. The gamma function is defined as

$$\Gamma(z) = \int_0^\infty t^{(z-1)} e^{-t} dt \qquad \text{Re}(z)>0$$

The gamma function is defined for all real and complex numbers except for negative integers. The gamma function can also be used to define factorials of negative numbers and non-integers, since

$$\Gamma(n+1) = n!$$

The CAS occasionally returns results including the gamma function, but there is no built-in function to calculate it. Here's a function to calculate gamma:

```
gamma(z)
Func
© Γ(Z) by Stirling's formula  ©M.Dave1
If real(floor(z))=z:Return when(z>Ø,(z-1)!,undef)
Local n,x,y
when(real(z)<Ø,1-z,z)→x
1Ø-min(floor(abs(x)),1Ø)→n
If  n≠Ø:x+n→x
approx(e^(¯x)*x^(x-Ø.5)*√(2*π)*polyEval({Ø.ØØØ83949872Ø872Ø9,¯5.17179Ø9Ø826Ø6E¯ØØ5,¯Ø.ØØØ
59216643735369,6.97281375836569E¯ØØ5,Ø.ØØØ784Ø39221720Ø7,¯Ø.ØØØ229472Ø936214,¯Ø.ØØ26813271
6Ø4938,Ø.ØØ347222222222222,Ø.Ø83333333333333,1},1/x)*when(n=Ø,1,Π(1/(x-k),k,1,n)))→y
approx(when(real(z)<Ø,π/(sin(z*when(sin(π)=Ø,π,18Ø)))/y,y))
EndFunc
```

This function works for all real and complex arguments. The accuracy is near full machine precision, except for *very* large negative arguments.

Since the gamma function is similar to the factorial function, the result overflows the calculator floating point range for relatively small arguments. For example, *gamma(z)* returns infinity for z>450. This limitation can be overcome by using a function that returns the natural log of the gamma function, instead of the gamma function itself. The log-gamma function is:

```
lngamma(z)
Func
© lnΓ(Z) by asymptotic series ©M.Dave1
If fPart(z)=Ø and z<1:Return undef
Local n,x,y
when(real(z)<Ø,1-z,z)→x
1Ø-min(floor(abs(x)),1Ø)→n
If n≠Ø:x+n→x
approx((x-Ø.5)*ln(x)-x+Ø.5*ln(2*π)+polyEval({¯Ø.ØØØ5952380952381,Ø.ØØØ79365Ø79365Ø79,¯Ø.Ø
Ø277777777777778,Ø.Ø83333333333333},x^(¯2))/x+when(n=Ø,Ø,ln(Π(1/(x-k),k,1,n))))→y
approx(when(real(z)<Ø,ln(π/(sin(z*when(sin(π)=Ø,π,18Ø))))-y,y))
EndFunc
```

The program author, Martin Daveluy, has these additional comments:

> *These two series use asymptotic series combined with the recurrence formula ( gamma(Z+1) = Z*gamma(Z) ) for Z<10 to keep full precision and the reflection formula ( gamma(Z)*gamma(1-Z) = pi/(sin(pi*Z)) ) to extend  domain to the entire complex plane. Note that the Gamma Stirling's fomula is obtained by this LnGamma formula. The Stirling's coefficients are obtained by collecting X power of the Maclaurin series for e^X ( 1+X+(X^2)/2!+... with X= LnGamma_asymptotic_series ) to reach higher precision.*

With a gamma function, it is possible to write a factorial function for non-integer and complex arguments:

```
factrl(xx)
func
©(x)factorial, complex & non-integer arguments
©1janØØ/dburkett@infinet.com
when(imag(xx)=Ø and fpart(xx)=Ø and xx≤45Ø,xx!,e^(math\lngamma(xx+1)))
Endfunc
```

If the input argument is a real integer less than 450, the result is found with the built-in factorial function is used, otherwise the log-gamma function is used. Note that *lngamma()* is installed in the *math* folder.

*(credit to Martin Daveluy)*

### [6.5] Round numbers to significant digits

You may need to round a number to a specified number of significant digits. For example, this can be useful when simulating arithmetic on a different processor with fewer significant digits, or estimating the effects of round-off error on function evaluation. The built-in function *round()* will not work, because it rounds to a specified number of digits after the decimal point. This function does it, though:

```
sigdig(x,n)
func
©(x,n) round x to n sig digits
©x is number, list, matrix
©n is 1 to 12
©13marØØ/dburkett@infinet.com
local s,xlm,k,j,n1,n2

if gettype(x)="NUM" then

format(approx(x),"s12")→s
return
expr(format(round(expr(left(s,instring(s,"ᴇ")-1)),n),"f"&string(exact(n-1)))&rig
ht(s,dim(s)-instring(s,"ᴇ")+1))

elseif gettype(x)="LIST" then
 dim(x)→n1
 newlist(n1)→xlm
 for k,1,n1
  sigdig(x[k],n)→xlm[k]
 endfor
 return xlm

elseif gettype(x)="MAT" then
 rowdim(x)→n1
 coldim(x)→n2
 newmat(n1,n2)→xlm
 for j,1,n1
  for k,1,n2
   sigdig(x[j,k],n)→xlm[j,k]
  endfor
 endfor
 return xlm

endif

Endfunc
```

Like the built-in *round()* function, *sigdig()* works on single numbers, lists or matrices. This is done by testing the type of the input argument *x*. If *x* is a number, it is rounded and returned. If *x* is a list or matrix, the individual elements are processed by a recursive calls to *sigdig().*

The actual rounding is performed by using *round()* on the mantissa of the argument. This process is simplifed by using *format()* to convert the input argument to scientific notation, which makes it easy to operate on the argument mantissa.

It would be an interesting challenge to modify this routine to work on complex numbers, in rectangular or polar format.

### [6.6] Linear regression through a fixed point

It can be useful or necessary to force a regression equation through a fixed point. For example, you might be modelling a system, and you know from the physical laws that the function must pass through (0,0). The 89/92+ built-in regression functions do not address this requirement. This function will find the fit coefficients for y = bx + a through a fixed point *(h,k).*

```
linreghk(xl,yl,h,k)
func
©return {b,a} for y=b*x+a through (h,k)
©14marØØ/dburkett@infinet.com
local s1,s2,s3

sum(xl^2)→s3

if h≠Ø then
sum(xl)→s2
(h*k*s2-k*s3-h^2*sum(yl)+h*sum(xl*yl))/(2*h*s2-s3-h^2*dim(xl))→s1
return {(k-s1)/h,s1}

else
return {sum(xl*yl)/s3,Ø}

endif

Endfunc
```

The code is a straightforward implementation of the regression equations from the book *Curve Fitting for Programmable Calculators*, William W. Kolb, Syntek, Inc., which is unfortunately out of print.

*xl* and *yl* are lists that specify the (x,y) data points to fit. *h* and *k* specify the point (h,k) through which to force the best-fit line. *linreghk()* returns a list that contains the coefficients {b,a}, where y = bx + a. Note that this list can be used directly with *polyeval()* to evalute the function at some point:

```
polyeval({b,a},x)
```

To force the fit curve through a specified a-value, use

```
linreghk(xl,yl,h,Ø)
```

To force the fit curve through the origin (0,0), use

```
linreghk(xl,yl,Ø,Ø)
```

Two different methods are needed, depending on whether or not h = 0. This routine does no error checking, but these criteria must be met:

1. The lists *xl* and *yl* must be the same length.
2. If h=0, then *xl* and *yl* must have at least two elements
3. If h is not equal to zero, then *xl* and *yl* must have at least three elements.
4. The function should be executed in Approx mode.

This data can be used to test the routine for a fit through the origin (0,0):

xl = {11,17,23,29}
yl = {15,23,31,39}

which returns {1.3483146067416,0}

This data can be used to test the routine for a fit through an arbitrary point:

xl = {100,200,300,400,500}
yl = {140,230,310,400,480}
h = 300
k = 310

which returns {0.85, 55}

This function can be used to find the unadjusted correlation coefficient for the curve fits:

```
corrhk(f1,xl,yl)
func
©(f1,xlist,ylist)return r^2 for y=f1(x)
©15mar00/dburkett@infinet.com

1-sum((seq(f1|x=xl[k],k,1,dim(xl))-yl)^2)/sum((seq(yl[k],k,1,dim(xl))-mean(yl))^
2)

Endfunc
```

For example, if the correlation equation coefficients are {b,a} as returned by *linreghk(),* use

```
corrhk(polyeval({b,a},x),xlist,ylist)
```

where *xlist* and *ylist* are the lists of data point coordinates.

### [6.7] Complex derivatives

The 89/92+ can find derivatives of functions in complex variables as well as those in real variables. Make sure that you specify that the variables are complex using the underscore character "_".

For example

```
d(1/(1-z_),z_)
```

returns

$$\frac{1}{(z\_-1)^{2}}$$

To put the result in terms of the real and imaginary components, use

```
d(f(z_),z_)|z_=a+bi
```

where 'i' is the complex unit operator. So,

```
d(1/z_,z_)|z_=a+bi
```

returns

$$\frac{-(a^2+b^2)}{(a^2+b^2)^2}+\frac{2\cdot a\cdot b}{(a^2+b^2)^2}i$$

If you fail to specify the variables as complex, you may not get the answer you expect. For example, the derivative of the complex conjugate function *conj()* is undefined, yet

```
d(conj(z),z)|z=a+bi
```

returns 1, which is *not* correct for complex *z*. This result comes about because the CAS assumes that *z* is a real variable, performs the differentiation, then substitutes *a+bi*.

Even though the rules for complex differentiation are the same as those for real differentiation, the criteria for complex differentiability are more stringent.

### [6.8] Convert trigonometric expressions to exponential format

Trigonometric expressions can be expressed as powers of the natural logarithm base *e*. Hyperbolic expressions can be easily converted to the equivalent exponential format with *expand().* For example:

```
expand(cosh(t))
```

returns $\frac{e^t}{2}+\frac{1}{2e^t}$

which is equivalent to the more common form $\frac{e^t+e^{-t}}{2}$

Other trigonometric functions can also be expressed in exponential form, but the results are more complicated. The principle is to use complex numbers for the function arguments, then eliminate the imaginary parts of the arguments. The procedure is:

1. Set the Complex Format mode to Polar.
2. Enter the trigonometric expression with the argument variables, using the "with" operator | to assign complex numbers to the arguments in rectangular coordinates.
3. Replace the imaginary components with zero to get the final result, using the "with" operator.

As an example, convert cos(x+y) to exponential format. Enter

```
cos(x+y)|x=a+bi and y=c+di
```

Note that x and y are replaced with complex numbers. The result is a sizable function of a, b, c and d. Next, enter

```
ans(1)|b=Ø and d=Ø
```

This eliminates the imaginary components of x and y. This also results in a big function, but it is only a function of *a* and *c*. The result is in the general form of $re^{i*theta}$, where *r* is the magnitude and *theta* is the angle of the result. *a* corresponds to the original *x*, and *c* corresponds to the original *y*. To make this substitution automatically, use this:

```
ans(1)|b=Ø and d=Ø and a=x and c=y
```

Setting constraints on *a* and *b* may result in a more simple expression.

*(Credit to Glenn Fisher)*

### [6.9] Convert floating-point numbers to exact fractions

The usual method to convert a floating-point number *n* to an exact fraction is to use *exact(n).* However, this function will only work for numbers smaller that the 14-digit precision of the 89/92+. For example, exact(1.234567890123456) results in 6172839450617/5000000000000, which is actually 1.23456789012; the ...3456 from the original number has been lost.

This program can be used to convert arbitrarily long floating point numbers to exact fractions.

```
stoexact(str)
Func
©Convert string argument to exact number

Local b,t,s

sign(expr(str))→s

if inString(str,"⁻")=1
right(str,dim(str)-1)→str

inString(str,".")→b

If b≠Ø then
 right(str,dim(str)-b)→t
 Return s*(exact(iPart(expr(str)))+exact(expr(t))/1Ø^(dim(t)))

else
 return exact(expr(str))
EndIf

EndFunc
```

The argument is passed as a string, for example

```
stoexact("1.234567890987654321")
```

returns 1234567890987654321/1000000000000000000.

This function works for positive and negative arguments, but does not support exponential notation.

*(credit for original idea & core code to Kenneth C. Arnold)*

### [6.10] Exact solutions to cubic and quartic equations

The 89/92+ functions *csolve()* and *czeros()* will not always return exact solutions to cubic and quartic equations. The routines *cubic()* and *quartic()* can be used to get the exact solutions.

*cubic():*

```
cubic(é,ý)
```

```
Func
©˜cubic(p,v) Roots of cubic equ
 p - cubic equ
 v - independent var

Local  p,r,q,ù,a,b,u,v,w
If  getType(é)≠"EXPR":Return  "p\cubic:arg must be a cubic eq"
p\coef(é,ý)→ù
If  dim(ù)≠4:Return  "p\cubic:arg must be a cubic eq"
string(cZeros(é,ý))→w
If  inString(w,".")=Ø and w≠"{}":Return  expr(w)
ù[2]/(ù[1])→p
ù[3]/(ù[1])→q
ù[4]/(ù[1])→r
(3*q-p^2)/3→a
(2*p^3-9*p*q+27*r)/27→b
(√(3*(4*a^3+27*b^2))-9*b)^(1/3)*2^(2/3)/(2*3^(2/3))→w
If getType(w)="NUM" Then
 If w=Ø Then
  If p=Ø Then
   ¯1→u
   1→v
  Else
   Ø→u
   Ø→v
  EndIf
 Else
  ω-a/(3*ω)→u
  ω+a/(3*ω)→v
 EndIf
Else
  ω-a/(3*ω)→u
  ω+a/(3*ω)→v
EndIf
{α-p/3,¯α/2+√(3)/2*𝑖*β-p/3,¯α/2-√(3)/2*𝑖*β-p/3,α=u,β=v,ω=w}
©Copyright 1998,1999 Glenn E. Fisher
EndFunc
```

*quartic():*

```
quartic(pp,ý)
Func
©˜quartic(p,v) Roots of 4th degree
© polynomial
© p - the polynomial
© v - variable

Local qq,ù,ú,r,d,e,y,j,k
If getType(ý)≠"VAR": Return "p\quartic:2nd arg must be a VAR"
p\coef(pp,ý)→ù
If dim(ù)≠5: Return "p\quartic:1st arg must be a 4th degree"
cZeros(pp,ý)→r
If inString(string(r),".")=Ø: Return r
ù/(ù[1])→ù
ý^3-ù[3]*ý^2+(ù[4]*ù[2]-4*ù[5])*ý+4*ù[3]*ù[5]-ù[4]^2-ù[2]^2*ù[5]→qq
p\cubic(qq,ý)→ú
p\evalrt(ú)→ú
Ø→k
For j,1,dim(ú)
 If real(ú[j])=ú[j] Then
  j→k
  Exit
 EndIf
EndFor
ú[k]→y
√(ù[2]^2/4-ù[3]+y)→r
If r=Ø Then
 √(3*ù[2]^2/4-2*ù[3]+2*√(y^2-4*ù[5]))→d
 √(3*ù[2]^2/4-2*ù[3]-2*√(y^2-4*ù[5]))→e
Else
 √(3*ù[2]^2/4-r^2-2*ù[3]+(4*ù[2]*ù[3]-8*ù[4]-ù[2]^3)/(4*r))→d
 √(3*ù[2]^2/4-r^2-2*ù[3]-(4*ù[2]*ù[3]-8*ù[4]-ù[2]^3)/(4*r))→e
EndIf
{¯ù[2]/4+r/2+d/2,¯ù[2]/4+r/2-d/2,¯ù[2]/4-r/2+e/2,¯ù[2]/4-r/2-e/2}
EndFunc
```

*coef():*

```
coef(è,ý)
Func
©˜coef(p,v) Make list of coefficents
 p - polynomial
 v - independent var

Local  ù,é,ì
If  getType(ý)≠"VAR":Return  "p\coef:2nd arg must be a VAR"
If inString(string(è),"=")>Ø Then
 left(è)→é
Else
 è→é
EndIf
é|ý=Ø→ù[1]
1→ì
Loop
 d(é,ý)→é
 If  getType(é)="NUM" and é=Ø:Exit
 ì+1→ì
 augment({(é|ý=Ø)/((ì-1)!)},ù)→ù
EndLoop
ù
EndFunc
```

*evalrt():*

```
evalrt(l)
Func
©˜evalRt(ls) Evaluate Roots
 ls - list of roots

Local  i,n,o,s,f
If  getType(l)≠"LIST": Return  "p\evalRt:Arg must be a LIST"
dim(l)→n
{}→o
Ø→f
For  i,1,n
 If inString(string(l[i]),"=")>Ø Then
  If f=Ø Then
   l[i]→s
   1→f
  Else
   s and l[i]→s
  EndIf
 Else
  augment(o,{l[i]})→o
 EndIf
EndFor
If f=Ø Then
 o
Else
 o|s
EndIf
EndFunc
```

*cubic()* is used to solve cubic (third-order) equations, and *quartic()* solves 4th-order equations. *coef()* and *evalrt()* are used by *cubic()* and *quartic()* to find the solutions. *evalrt()* can also be used on the results from *cubic()* and *quartic()*, as described below, to expand the solutions.

All of these routines must be in the same folder, called *p*. The mode must be set to Auto, and *not* to Exact or Approx. Set Complex format to Rectangular or Polar. To help prevent "Memory" errors, these routines should be the only variables in the folder, and there should be no other variables in the folder. These routines can take a long time to return the result.

The input to both *cubic()* and *quartic()* is a polynomial in *x* or *z*. The output is a list of the solutions. For example,

| | | |
|---|---|---|
| cubic($x^3$-12$x^2$-15x+26,x) | returns | {-2 1 13} |
| quartic($z^4$-2$z^3$-13$z^2$+38z-24,z) | returns | {-4 1 2 3} |

The advantage of using *cubic()* and *quartic()* is that they return exact answers, which *czeros()* or *csolve()* don't always do. For example,

czeros($x^3$-x+1,x)

returns

{-1.32472  0.662359 - .56228i   0.662359+.56228i}

while

cubic($x^3$-x+1,x)

returns a list with these solution elements:

$$\alpha$$

$$\frac{-\alpha}{2}+\frac{\beta\cdot\sqrt{3}}{2}\cdot i$$

$$\frac{-\alpha}{2}-\frac{\beta\cdot\sqrt{3}}{2}\cdot i$$

$$a=\omega+\frac{1}{3\cdot\omega}$$

$$\beta=\omega-\frac{1}{3\cdot\omega}$$

$$\omega=\frac{(-3\cdot(\sqrt{69}-9))^{\frac{1}{3}}\cdot 2^{\frac{2}{3}}}{12}+\frac{(9-\sqrt{69})^{\frac{1}{3}}\cdot 3^{\frac{5}{6}}\cdot 2^{\frac{2}{3}}}{12}\cdot i$$

The first three elements are the solutions to the cubic polynomial. The last three elements define the variable substitutions used in the solutions.  This format reduces the size of the output list, and makes the solutions more intelligible. The function *evalrt()* automatically applies these substitutions. *evalrt()* is called with the solution list output from *cubic()*, with these results returned as a list:

$$\frac{3^{\frac{2}{3}}\cdot 2^{\frac{1}{3}}}{6\cdot(9-\sqrt{69})^{\frac{1}{3}}}+\frac{(-3\cdot(\sqrt{69}-9))^{\frac{1}{3}}\cdot 2^{\frac{2}{3}}}{12}+\left(\frac{(9-\sqrt{69})^{\frac{1}{3}}\cdot 3^{\frac{5}{6}}\cdot 2^{\frac{2}{3}}}{12}-\frac{3^{\frac{1}{6}}\cdot 2^{\frac{1}{3}}}{2\cdot(9-\sqrt{69})^{\frac{1}{3}}}\right)\cdot i$$

$$\frac{-\left((9-\sqrt{69})^{\frac{2}{3}}\cdot 2^{\frac{1}{3}}+2\cdot 3^{\frac{1}{3}}\right)\cdot 6^{\frac{1}{3}}}{6\cdot(9-\sqrt{69})^{\frac{1}{3}}}$$

$$\frac{\left((9-\sqrt{69})^{\frac{2}{3}}\cdot 2^{\frac{1}{3}}+2\cdot 3^{\frac{1}{3}}\right)\cdot 6^{\frac{1}{3}}}{12\cdot(6-\sqrt{69})^{\frac{1}{3}}}-\frac{\left((-3\cdot(\sqrt{69}-9))^{\frac{2}{3}}\cdot 2^{\frac{1}{3}}-6\right)\cdot 3^{\frac{1}{6}}\cdot 2^{\frac{1}{3}}}{12\cdot(9-\sqrt{69})^{\frac{1}{3}}}\cdot i$$

These solutions can be verified by substituting them back into the original polynomial.

For another example, try quartic($x^4$-x+1,x). The results are quite long, and not shown here. Since they involve inverse trigonometric functions, they cannot be checked by substituting them into the original polynomial. However, if they are substituted into the original polynomial and approximate results are found, those results are zero to within machine precision.

If the polynomial to be solved has complex coefficients, you must specify that the argument variable is complex with the underscore character "_", or the complex solutions will not be returned. For example, solve the polynomial generated by

$$(x+2)\cdot(x-3)\cdot(x-7i)$$

which obviously has roots of -2, 3 and 7i. This expression expands to

$$x(x-3)(x+2)-7(x-3)(x+2)i$$

If you call cubic like this:

```
cubic(x*(x-3)*(x+2)-7*(x-3)*(x+2)*i,x)
```

then the returned solutions are {-2,3}: the complex solution of 7i is not returned. However, if *cubic()* is properly called with the underscore, to indicate that *x* is complex, like this

```
cubic(x_*(x_-3)*(x_+2)-7*(x_-3)*(x_+2)*i,x_)
```

then the correct solutions of {-2, 3, 7i} are returned.

*cubic()* and *quartic()* first try to find an exact solution with *czeros().* If an exact solution is not returned, they use Cardano's formula to find the solutions. One reference which describes Cardano's formula is the *CRC Standard Mathematical Tables and Formula*, 30th edition, Daniel Zwillinger, Editor-in-Chief.

*(Contributor declines credit. Credit also to Glenn Fisher and Pini Fabrizio.)*

### [6.11] Rounding floating point numbers

To round a floating point number *n* for further use in calculations, use

```
expr(format(n,fcode))
```

where *fcode* is a format string. To round a number to a specific number of fractional decimal digits, use *fcode* of "Fd", where 'd' is the number of digits. For example, for three fractional digits, use

```
expr(format(1.23456789,"F3"))
```

which returns 1.235

To round a number to a given number of significant digits, use *fcode* of "Sd", where (d + 1) is the number of significant digits. For example, for 6 signficant digits, use

```
expr(format(1.23456789E1Ø,"S5"))
```

which returns 1.23457E10

This method works by using the *format()* function to do the rounding, then using *expr()* to convert the string result back to a number. The distinction between fractional digits and significant digits is made by using "Fd" for fixed format, or "Sd" for scientific format.

*(Credit to Andrew Cacovean)*

### [6.12] Find faster numerical solutions for polynomials

As shown in tip [11.5], *nsolve()* is accurate and reliable in solving polynomial equations, but the speed leaves something to be desired. This tip shows a way to find the solution faster. This method requires that the equation to be solved is a polynomial, and that an estimating function can be found. This estimating function must be a polynomial, as well.

The basic method is to replace *nsolve()* with a TI Basic program that uses Newton's method, coupled with an accurate estimating function. Further speed improvements are possible since you control the accuracy of the solution.

The function is called *fipoly(),* and here is the code:

```
fipoly(clist,fguess,yval,yemax,yetype)
func
©Fast inverse polynomial solver
©26 nov 99 dburkett@infinet.com
©Find x, given f(x)
©clist: list of polynomial coefficients
©fguess: list of guess generating polynomial coefficients
©yval: point at which to find 'x'
©yemax: max error in 'y'
©yetype: "rel": yemax is relative error; "abs": yemax is absolute error.

local fd,dm,xg,yerr,n,erstr,nm

©Set maximum iterations & error string
3Ø→nm
"fipoly iterations"→erstr

©Find list of derivative coefficients
dim(clist)→dm
seq(clist[k]*(dm-k),k,1,dm-1)→fd

©Find first guess and absolute error
polyeval(fguess,yval)→xg
polyeval(clist,xg)-yval→yerr

©Loop to find solution 'x'
Ø→n

if yetype="abs" then
 while abs(yerr)>yemax
  xg-(polyeval(clist,xg)-yval)/polyeval(fd,xg)→xg
  polyeval(clist,xg)-yval→yerr
  n+1→n
  if n=nm:return erstr
 endwhile
else
 yerr/yval→yerr
 while abs(yerr)>yemax
  xg-(polyeval(clist,xg)-yval)/polyeval(fd,xg)→xg
  (polyeval(clist,xg)-yval)/yval→yerr
  n+1→n
  if n=nm:return erstr
 endwhile

endif

xg

Endfunc
```

Again, this routine will only work if your function to be solved is a polynomial, and you can find a fairly accurate estimating function for the solution. (If you can find a *very* accurate estimating function, you don't need this routine at all!)

The function parameters are

| | |
|---|---|
| clist | List of coefficients for the polynomial that is to be solved. |
| fguess | List of coefficients of the estimating (guess) polynomial |
| yval | The point at which to solve for x |
| yemax | The maximum desired y-error at the solution for x, must be >0 |
| yetype | A string that specifies whether yemax is absolute error or relative error:<br>"abs" means absolute error<br>"rel" means relative error |

*fipoly()* returns the solution as a numeric value, if it can find one. If it cannot find a solution, it returns the string "fipoly iterations". If you call *fipoly()* from another program, that program can use *gettype()* to detect the error, like this:

```
fipoly(...)→x
if getype(x)≠"NUM" then
 {handle error here}
endif
{otherwise proceed}
```

With *fipoly(),* you can specify the y-error *yemax* as either a relative or absolute error. If $y_a$ is the approximate value and y is the actual value, then

$$\text{absolute error} = |y - ya| \qquad \text{relative error} = \left|\frac{y-ya}{y}\right|$$

You will usually want to use the relative error, because this is the same as specifying the number of significant digits. For example, if you specify a relative error of 0.0001, and the y-values are on the order of 1000, then *fipoly()* will stop when the y-error is less than 0.1, giving you 4 significant digits in the answer.

However, suppose you specify an absolute error of 1E-12, and the y-values are on the order of 1000 as above. In this case, *fipoly()* will try to find a solution to an accuracy in y of 1E-12, but the y-values only have a resolution of 1E-10. *fipoly()* won't be able to do this, and will return the error message instead of the answer.

As an example, I'll use the same gamma function approximation function from tip [11.5]. The function to be solved is

$$y = a + bx + cx^2 + dx^3 + ex^4 + fx^5 + gx^6 + hx^7 + ix^8$$

where these coefficients are saved in a list variable called *fclist*:

| | | |
|---|---|---|
| a = 4.44240042385 | b = -10.1483412133 | c = 13.4835814713 |
| d = -11.0699337662 | e = 6.01503554007 | f = -2.15531523837 |
| g = 0.494033458314 | h = -.0656632350273 | i = 0.00388944540448 |

Using curve fitting software for a PC, I found this estimating function:

$$x = p + qy + ry^2 + sy^3 + ty^4 + uy^5$$

where these coefficients are saved in a variable called *fglist*:

| | | |
|---|---|---|
| p = -788.977246657 | q = 3506.8808748 | r = -6213.31596202 |

s = 5493.68334077 t = -2422.15013853 u = 425.883370029

So, to find x when y = 1.5, with a relative error of 1E-8, the function call looks like this:

fipoly(fclist,fglist,1.5,1E-8,"rel") which returns x = 2.6627...

Using the same test cases as in tip [11.5], the table below shows the execution times and errors in *x* for various maximum *y* error limits, for both the relative and absolute error conditions.

| yemax | "abs" max x-error | "rel" max x-error | "abs" mean execution time, sec | "rel" mean execution time, sec |
|---|---|---|---|---|
| 1 E-1 | 3.03 E-2 | 3.03 E-2 | 0.70 | 0.72 |
| 1 E-2 | 3.03 E-2 | 3.03 E-2 | 0.70 | 0.71 |
| 1 E-3 | 6.56 E-3 | 6.56 E-3 | 0.90 | 0.92 |
| 1 E-4 | 6.96 E-4 | 6.96 E-4 | 0.95 | 0.99 |
| 1 E-5 | 1.74 E-5 | 1.74 E-5 | 1.12 | 1.15 |
| 1 E-6 | 3.19 E-6 | 3.19 E-6 | 1.16 | 1.20 |
| 1 E-7 | 2.82 E-6 | 3.21 E-7 | 1.23 | 1.30 |
| 1 E-8 | 1.35 E-8 | 1.35 E-8 | 1.27 | 1.32 |
| 1 E-9 | 6.67 E-10 | 6.67 E-10 | 1.30 | 1.33 |
| 1 E-10 | 6.67 E-10 | 6.67 E-10 | 1.38 | 1.43 |
| 1 E-11 | 6.67 E-10 | 5.84 E-10 | 1.41 | 1.45 |
| 1 E-12 | 5.84 E-10 | 5.84 E-10 | 1.77 | 1.83 |

There is little point to setting the error tolerance to less than 1E-12, since the 89/92+ only use 14 significant digits for floating point numbers and calculations. For this function, we don't gain much by setting the error limit to less than 1E-9.

Note that this program is much faster than using *nsolve():* compare these execution times of about 1.3 seconds, to those of about 4 seconds in tip [11.5].

The code is straightforward. The variable *nm* is the maximum number of iterations that *fipoly()* will execute to try to find a solution. It is set to 30, but this is higher than needed in almost all cases. If Newton's method can find an answer at all, it can find it very quickly. However, I set *nm* to 30 so that it will be more likely to return a solution if a poor estimating function is used.

I use separate loops to handle the relative and absolute error cases, because this runs a little faster than using a single loop and testing for the type of error each loop pass.

### [6.13] Find coefficients of determination for all regression equations

The 89/92+ can fit 10 regression equations, but do not find the coeffcient of determination $r^2$ for all the equations. However, $r^2$ is defined for any regression equation, and these two functions calculate it:

```
r2coef(lx,ly)
func
©Find coefficient of determination r^2, no adjustment for DOF
©lx is list of x-data points
©ly is list of y-data points
©24 nov 99/dburkett@infinet.com

1-sum((regeq(lx)-ly)^2)/sum((ly-mean(ly))^2)
```

```
Endfunc


r2coefdf(lx,ly)
func
©Find coefficient of determination r^2, adjusted for DOF
©lx is list of x-data points
©ly is list of y-data points
©24 nov 99/dburkett@infinet.com
local n
dim(lx)→n

1-((n-1)*sum((regeq(lx)-ly)^2))/((n-dim(regcoef)-1)*sum((ly-mean(ly))^2))

Endfunc
```

Note that neither of these functions will account for weighting of the data points.

Both functions should be run in Approx mode. Both take the list of x-data and y-data values in the lists *lx* and *ly*. Be sure to run the regression command (*CubicReg*, *ExpReg*, etc) before using these functions, because they depend on the system variables *regeq* and *regcoef* being up-to-date.

*regeq* is a function that is updated by the regression command. It is the regression equation itself, and I use it in both routines to calculate needed values. *regcoef* is the list of regression equation coefficients, and I use it in *r2coefdf()* to find the degrees of freedom.

The coefficient of determination, without adjustment for degrees of freedom, is defined as

$$r2 = 1 - \frac{SSE}{SSM}$$

SSE is the sum of the squares of the error, defined as

$$SSE = \sum_{i=1}^{n} (\hat{y}_i - y_i)^2$$

SSM is the sum of squares about the mean, defined as

$$SSM = \sum_{i=1}^{n} (y_i - \bar{y})^2$$

Also,

n is the number of data points
$y_i$ are the y-data values
$\hat{y}_i = f(x_i)$ are the estimated y-values, that is, the regression equation evaluated at each x-data value
$\bar{y}$ is the mean of the y-data values

It is a simple matter to adjust $r^2$ for degrees of freedom:

$$r2 = 1 - \frac{(n-1)\cdot SSE}{(DOF-1)\cdot SSM}$$

Here, DOF is the degrees of freedom, defined as

$$DOF = n - m$$

where *m* is the number of coefficients in the equation. For the simple linear equation y = ax + b, m = 2, because we have two coefficients *a* and *b*.

It is appropriate to use $r^2$ without DOF adjustment when the data to be fit has no experimental errors. For example, this is the case if you are fitting a curve to data generated by some other mathematical function. However, the DOF adjusted $r^2$ should be used when the y-data values include experimental error.

### [6.14] Use *norm()* to find root mean square (RMS) statistic for matrices

The root mean square statistic is used in statistics. I prefer to use it over other statistics for comparing curve fits and interpolation errors. RMS is defined as

$$\mathrm{RMS} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(x_i^2\right)}$$

The 89/92+ *norm()* function is very close to this:

$$\mathrm{norm}([a,b,c,...]) = \sqrt{a^2+b^2+c^2+...}$$

so we can rewrite the RMS equation and use the *norm()* function to find RMS:

$$\mathrm{RMS} = \frac{1}{\sqrt{n}}[\mathrm{norm}(\mathrm{dmatrix})]$$

where *dmatrix* is the matrix of data values, and *n* is the size of the matrix. This function performs the calculation:

```
rms(m)
func
if gettype(m)="LIST":list▸mat(m)→m
norm(m)/(√(max(dim(m))))
Endfunc
```

The input variable *m* may be a list or a matrix. If *m* is a matrix, it may be either a single-row or a single-column matrix. This is handled by using the maximum value returned by *dim()* as the value of *n*.

### [6.15] Convert equations between rectangular and polar coordinates

It can be useful to convert equations between polar and rectangular coordinates, and the 89/92+ have functions that make this particularly easy. These two programs, by Alex Astashyn, perform the conversion. *p2c()* converts from polar to rectangular, and *c2p()* converts from rectangular to polar.

```
c2p(eq)
func
©Convert 'eq' in rectangular coordinates x,y to polar coordinates r,θ
solve(eq,r)|x=P▸Rx(r,θ) and y=P▸Ry(r,θ)
EndFunc

p2c(eq)
Func
©Convert 'eq' in polar coordinates r,θ to
rectangular coordinates x,y.
solve(eq,y)|θ=R▸Pθ(x,y) and r=R▸Pr(x,y)
EndFunc
```

These programs use built-in conversion functions to perform these general coordinate transformations on the equation:

$$x = r \cdot \cos(\theta)$$
$$y = r \cdot \sin(\theta)$$

$$r = \pm \sqrt{x^2 + y^2}$$

$$\theta = \tan^{-1}\left(\frac{y}{x}\right)$$

The transformation equations might not return the results you expect, because the last two equations do not uniquely define *r* and $\theta$.

For example, to find the polar coordinate equation for the straight line y = ax + b, use

```
c2p(y=a*x+b)
```

which returns

$$r = \frac{-b}{a\cdot\cos(\theta)-\sin(\theta)}$$

To convert the polar equation for a circle with radius = 4 to rectangular coordinates, use

```
p2c(r=4)
```

which returns

$$x^2 - 16 \le 0 \text{ and } y = -\sqrt{16-x^2} \text{ or } x^2 - 16 \le 0 \text{ and } y = \sqrt{16-x^2}$$

This shows a general issue when converting from one coordinate system to the other, which is that extraneous solutions may be introduced. The only general solution is to check each answer.

*(Credit to Alex Astashyn)*

### [6.16] Transpose operator and dot product find adjoint and complex scalar product

I couldn't have said it better myself:

*"Actually, it is worth noting that the transpose operator "T" works as the adjoint (complex conjugate transposed).*

*For example, [1,i;1,2i]$^T$ is [1,1;-i,-2i].*

*In the same context, the scalar product dotP() works correctly as a complex scalar product. It is linear in the first argument, antilinear in the second. For example dotP([1,1],[1,i]) is 1-i."*

*(Credit to fabrizio)*

### [6.17] Use dd.mmssss format to convert angles faster

Surveyors and civil engineers commonly measure angles in degrees, minutes, seconds and fractions of a second. The 89/92+ can convert these angles to decimal angles: just enter the angle on the command line, and enter it:

12°34'56" [ENTER]

gives the result 12.582222° in the Degree angle mode, and 0.219601 rad in Radian angle mode.

However, to enter this angle requires these keystrokes:

1, 2, [2nd][d], 3, 4, [2nd][b], 5, 6, [2nd][l], [ENTER]

That is an additional six keystrokes to enter the degrees, minutes and seconds characters. This is time-consuming if you need to do it a lot, and you either have to remember the [2nd][d] and [2nd][l] shortcuts, or look them up.

An alternative method to enter angles is the format *dd.mmssss*. The angle is entered as a floating point number, where *dd* is degrees, *mm* is minutes, and *ssss* is seconds. For example,

12.345678 is equivalent to 12° 34' 56.78"

This angle format is used on the HP48/49 series of calculators, among other machines. Since the 89/92+ do not directly support this angle format, you need a conversion routine to convert the entered angle to decimal degrees or radians. The routine, called *dms(),* is:

```
dms(a1)
func
©Convert angle in D.MS format to decimal degrees or radians.
©By Doug Burkett & anonymous poster
©Input angle a1 is in format dd.mmsss, 12.345678 = 12°34'56.78"
©Returns result in degrees in Degree angle mode; in radians in Radian mode.

((100*fpart(100*a1))/3600+(ipart(100*fpart(a1))/60)+ipart(a1))°

Endfunc
```

This routine works for positive and negative angles, regardless of the display exponent mode setting. The correct result is returned depending on the current Angle mode, either radians or degrees. This is accomplished by ending the function equation with the ° symbol. I thank the anonymous poster for pointing this out.

You might think that the inverse conversion, from decimal degrees or radians, to degrees, minutes and seconds, could be done with the ▸DMS instruction. This is true if you only want to *see* the conversion result. For example

12.5▸DMS returns 12°30'

This is not in the required DMS format of 12.30. This routine converts decimal degrees or radians to the DMS format:

```
dmsi(a1)
func
©Convert angle in decimal degrees or radians to D.MS format
```

```
©By Doug Burkett & anonymous poster
©Angle a1 is returned in format dd.mmsss, 12.345678 = 12°34'56.78"
©Assumes angle 'a1' is degrees if angle mode is Degree, or radians if the mode
is Radian.

if getmode("angle")="RADIAN":a1*57.2957795130820→a1

ipart(a1)+.01*(ipart(60*fpart(a1)))+.006*fpart(60*fpart(a1))

Endfunc
```

In summary:

*dms()* converts an angle in DMS format to degrees in Degree mode, or radians in Radian mode.

*dmsi()* converts an angle to DMS format. The angle is assumed to be degrees in Degree mode, and radians in Radian mode.

### [6.18] Use *iPart()* and *int()* effectively

*iPart()* and *int()* return the same results for positive numbers, but operate differently for negative numbers. Remember that *int()* is identical to *floor(),* so it returns the greatest integer that is less than or equal to the argument. So,

int(4.2) = iPart(4.2) = 4

but

int(-4.2) = -5 and iPart(-4.2) = -4

If you are trying to find the integer part of any number, use *iPart().*

### [6.19] Sub-divide integration range to improve accuracy

You can improve the accuracy of numerical integration by taking advantage of the fact that the 89/92+ numerical integrator is more accurate over small intervals. For example, consider this function:

$$Q(x) = \frac{1}{\sqrt{2\pi}} \int_{x}^{\infty} e^{-\frac{t^2}{2}} dt$$

This is actually the integral for the the complement of the cumulative normal probability distribution function. If you just integrate this function as shown, for x = 1,

```
1/(√(2·π))·∫(e^(-t^2/2),t,1,∞) = 0.1586 5525 3456 96
```

which has an error of about 4.745E-10. To get a more accurate result, find integrals over smaller ranges that cover the entire integration limits range, and sum these to get the complete integral. The table below shows the individual integrals for some ranges I chose.

| Integration limits | Integral |
|---|---|
| t = 1 to t = 2 | 0.1359 0512 1983 28 |
| t = 2 to t = 3 | 0.0214 0023 3916 549 |
| t = 3 to t = 4 | 0.0013 1822 6789 7969 |
| t = 4 to t = 5 | 3.1384 5902 6124 E-5 |
| t = 5 to t = 6 | 3.8566 4984 2341 4 E-7 |
| t = 6 to t = 7 | 9.8530 7832 4938 2 E-10 |
| t = 7 to t = 8 | 1.2791 9044 7828 2 E-12 |
| t = 8 to t = 9 | 6.2198 3198 5865 6 E-16 |

Summing these individual integrals gives a result of 0.1586 5525 3931 46, which has an error of zero, to 14 digits.

Note that I stopped taking integrals when the individual integral is less than 1E-14, since further, smaller integrals will not contribute to the sum.

This method can be used with the ∫ operator, or with the *nInt()* function.

The function *nintx(),* shown below, automates this process.

```
nintx(ffx,xv,xx1,xx2,nx)
func
©Numerical integrator with summed subintervals
©dburkett@infinet.com 6 nov 99
©ffx: function to integrate
©xv: variable of integration
©xx1,xx2: lower and upper integration limits
©nx: number of subintervals

local dx

(xx2-xx1)/nx→dx

sum(seq(nint(ffx,xv,xx1+i*dx,xx1+(i+1)*dx),i,Ø,nx-1))

endfunc
```

For example, the call nintx(tan(x),x,0,1.5707,5) integrates tan(x) from x=0 to x=1.5707, with 5 subintervals.

The amount of improvement depends on the function. The built-in *nint()* function has an error of -29.7E-12 for the tan(x) example above. *nintx()* with 10 intervals has an error of about 13E-12.

### [6.20] Generating random numbers

The *rand()* function can only be used to generate random integers in the intervals [1,n] or [-n, -1]. Use this to generate random integers over any range [nl,nh]:

```
rand(nh-nl+1)+nl-1
```

If *rand()* is called with no arguments, it returns a random floating point number between 0 and 1. To generate uniformly distributed random numbers over the range [fl,fh], use this:

```
(fh-fl)*rand()+fl
```

You may want to use *RandSeed* to reset or initialize the random sequence. Also note that the *randNorm()* function returns normally distributed random numbers.

### [6.21] Evaluating polynomials

You don't need to explicitly calculate a polynomial like this:

```
2*x^3 + 3*x^2 + 4*x - 7 → result
```

Instead, TI BASIC has a *polyEval()* function that uses less ROM space:

```
polyEval({2,3,4,-7},x) → result
```

I don't have any specific timing results, but this method is very fast. You can use this method whenever you have a sequence of powers of any function. Some examples:

| | | |
|---|---|---|
| $\frac{4}{x^3}-\frac{6}{x^2}+\frac{8}{x}-3$ | use | `polyEval({4,-6,8,-3},1/x)` |
| $\frac{4}{[\ln(x)]^3}-\frac{6}{[\ln(x)]^2}+\frac{8}{\ln(x)}-3$ | use | `polyEval({4,-6,8,-3},1/ln(x))` |
| $4[\sin(x)]^3 - 6[\sin(x)]^2 - 3$ | use | `polyEval({4,-6,Ø,-3},sin(x))` |

### [6.22] Linear Interpolation

Linear interpolation is the least sophisticated, least accurate interpolation method. However, it has the advantages that it is fast, and often accurate enough if the table values are closely spaced. Given a table of function values like this:

| x1 | y1 |
|---|---|
| x | y |
| x2 | y2 |

where *x1*, *y1*, *x2* and *y2* are given, the problem is to estimate *y* for some value of *x*. In general,

$$y = y1 + (y2 - y1)\left[\frac{x-x1}{x2-x1}\right]$$

which can be derived by finding the equation for the line that passes through (x1,y1) and (x2,y2).

If you interpolate often, it is worth defining this as a function. At the command line:

```
define interp(xx1,yy1,xx2,yy2,x)=yy1+(yy2-yy1)*((x-xx1)/(xx2-xx1))
```

or in the program editor:

```
interp(xx1,yy1,xx2,yy2,x)
Func
©Linear interpolation
yy1+(yy2-yy1)*((x-xx1)/(xx2-xx1))
EndFunc
```

Note that I use variable names of the form *yy1* to avoid contention with the built-in function variables *y1* and *y2*. If xx1 = 1, yy1 = 1, xx2 = 2 and yy2 = 4, either of these routines returns y = 2.5 for x = 1.5.

Either of these methods are fast and work well. However, you have to remember the order of the input variables. One way to avoid this is to use the Numeric Solver. This program automates the process:

```
linterp()
Prgm
©Linear interpolation with numeric solver

©Delete equation variables
delvar x,y,xx1,xx2,yy1,yy2

©Save interpolation equation
y=((yy1-yy2)*x+xx1*yy2-xx2*yy1)/(xx1-xx2)→eqn

©Start the numeric solver
setMode("Split 1 App","Numeric Solver")
EndPrgm
```

This program works by saving the interpolation equation to the system variable *eqn*, which is used by the numeric solver. All the equation variables are deleted first, otherwise any current values will be substituted into the equation and it will be simplified, which won't give the correct equation to solve. Finally, the numeric solver is started with the last program line.

To use this program, enter *linterp()* at the command line, then press ENTER at the solver *eqn:* prompt. The prompts for the six variables are shown. Enter the required values for *x*, *xx1*, *xx2*, *yy1* and *yy2*, then solve for *y*.

Another advantage of this program is that it is easy to find *x*, given *y*. This process is sometimes called inverse interpolation. You can also use the *interp()* functions above for inverse interpolation: just enter the y-values as x-values, then solve for *y* as usual, which will actually be the interpolated value for *x*.

### [6.23] Step-by-step programs

These sites have programs that solve various problems step-by-step, that is, the program shows each of the steps used to solve a problem. I have not tried these programs.

Oliver Miclo's ti-cas site: *http://www.ti-cas.org/*

- Compute derivative: stepder.89g (TI-89) or stepder.9XG (TI-92+)
- Solution of linear systems: *invm()*

TIcalc site: *http://www.ticalc.org/pub/89/basic/math/*

- Solve Diophant equations: diophant.zip
- Apply the Euclid algorithm to two numbers: euclide2.zip
- Solve a 3x3 augmented matrix with Gauss-Jordan elimination: matsol.zip

TIcalc site: *http://www.ticalc.org/pub/92/basic/math/*

- Solve *n* equations in *n* unknowns using Gauss-Jordan elimination: srref.zip

### [6.24] Fast Fibonacci Numbers

The Fibonacci numbers are defined by this recurrence relation:

$F_1 = 1, \qquad F_2 = 1, \qquad F_{n+2} = F_n + F_{n+1}$

This formula can be used to find the nth Fibonacci number:

$$F_n = \frac{1}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right]$$

However, the 89/92+ cannot use this formula for large *n*, because the CAS cannot simplify the result. Finding the numbers with a TI Basic program in recursion is quite slow, and limits *n* to about 255. However, this function can find Fibonacci numbers for large *n*:

```
Define fib(n)=([[1,1][1,Ø]]^(n-1))[1,1]
```

*(Credit to Alex Astashyn)*

### [6.25] Polar and rectangular coordinate conversions

The 89/92+ provide a variety of methods to convert point or vector coordinates between polar and rectangular formats.

You can use the Vector Format setting in the Mode menu to perform conversions automatically from the command line. If the Vector Format is set to RECTANGULAR, then coordinates entered in polar format are converted to rectangular coordinates when the vector is entered. For example, if the mode is RECTANGULAR and `[√(2),∠π/4]` in polar coordinates is entered, then [1,1] is returned. If the Vector Format is set to CYLINDRICAL or SPHERICAL and [1,1] in rectangular coordinates is entered, then `[√(2),∠π/4]` is returned.

You can also use the conversion functions `▸Polar` and `▸Rect` to convert vector coordinates. For example, `[1,1]▸Polar` returns `[√(2),∠π/4]`, and `[√(2),∠π/4]▸Rect` returns [1,1]. However, note that `▸Polar` and `▸Rect` are called "display-format" instructions. They only affect the display of the vector: they do not really convert the vectors.

Finally, these functions can be used to return just one part of a converted vector:

| | |
|---|---|
| `P▸Rx(r,θ)` | Return x-coordinate of polar vector argument |
| `P▸Ry(r,θ)` | Return y-coordinate of polar vector argument |
| `R▸Pr(x,y)` | Return magnitude r of rectangular vector argument |
| `R▸Pθ(x,y)` | Return angle $\theta$ of rectangular vector argument |

These functions can be used to write user functions which, unlike `▸Polar` and `▸Rect`, *do* actually convert the input vector. These functions are:

```
polar(v)
```

```
func
©Convert vector v to polar
[R▸Pr(v[1,1],v[1,2]),R▸Pθ(v[1,1],v[1,2])]
EndFunc

rect(v)
func
©Convert vector v to rectangular
[P▸Rx(v[1,1],v[1,2]),P▸Ry(v[1,1],v[1,2])]
EndFunc
```

`polar([1,1])` returns `[[√(2),π/4]]`. `rect([[√(2),π/4]])` returns [1,1]. Note that these routines assume the current Angle Mode setting. If the angle mode is radians, then the angles are in radians. If you or your application expect the arguments or results to be in degrees or radians, set the proper mode.

*(Credit to Sam Jordan for prompting my write-up; code (and bugs) are mine)*

### [6.26] Accurate numerical derivatives with nDeriv() and Ridder's method

This tip is lengthy. It is divided into these sections:

- *Optimum results from nDeriv.* How to use the built-in *nDeriv()* function to get the most accurate results*,* with a function called *nder2().*
- *More accurate results with Ridders' method.* Shows a method (Ridders') that gives much better results than *nDeriv(),* with a program called *nder1().*
- *Diagnosing nder1() with nder1p().* Shows how to fix errors that might occur with the Ridders' method program*.*
- *General comments on numerical differentiation.* General concerns with any numerical differentiation method.
- *More comments on nder1() and Ridders' method*. Lengthy discussion of *nder1()* results, performance, and also some interesting behavior of the built-in *nDeriv()* function.

In general, the most accurate method to find a numerical derivative is to use the CAS to find the symbolic derivative, then evaluate the derivative at the point of interest. For example, to find the numeric derivative of tan(x), where x = $\pi$/2.01 = 1.562981..., find

$$\frac{d}{dx}\tan(x) = \frac{1}{(\cos(x))^2} = 16374.241898666$$

This method fails when the CAS cannot find a symbolic expression for the derivative, for example, for a complicated user function. The 89/92+ provide two built-in functions for finding numerical derivatives: *avgRC()* and *nDeriv(). avgRC()* uses the forward difference to approximate the derivative at *x*:

$$\text{avgRC}(f(x), x, h) = \frac{f(x+h)-f(x)}{h}$$

and *nDeriv()* uses the central difference to approximate the derivative:

$$\text{nDeriv}(f(x), x, h) = \frac{f(x+h)-f(x-h)}{2h}$$

The built-in functions *avgRC()* and *nDeriv()* are fast, but they may not be accurate enough for some applications. The *avgRC()* function can return, at best, an accuracy on the order of $e_m^{1/2}$, where $e_m$ is the machine accuracy. Since the 89/92+ uses 14 decimal digits, $e_m$ = 1E-14, so the best result we can expect is about 1E-7. For *nDeriv(),* the best accuracy will be about $e_m^{2/3}$, or about 5E-10. Neither of these accuracies reach the 12-digit accuracy of which the 89/92+ is capable. It is also quite possible that the actual error will be much worse. Our goal is to develop a routine which can calculate derivatives with an accuracy near the full displayed resolution of the 89/92+.

Since the title of this tip is *accurate* numeric derivatives, I won't further consider *avgRC().* Instead, I will show how to get the best accuracy from *nDeriv(),* and also present code for an alternative method that does even better.

### *Optimum results from nDeriv(): nder2()*

*nDeriv()* finds the central difference over an interval *h*, where *h* defaults to 0.001. Since the limit of the *nDeriv()* formula *is* the derivative, it makes sense that the smaller we make *h*, the better the derivative result. Unfortunately, round-off errors dominate the result for too-small values of *h*. Therefore, there is an optimum value of *h* for a given function and evaluation point. An obvious strategy is to evaluate the formula with increasingly smaller values of *h*, until the absolute value of successive differences in f'(x) begins to increase. This program, *nder2(),* shows this idea:

```
nder2(ff,xx)
func
©("f",x) find best f'(x) at x with central-difference formula
©6jun00 dburkett@infinet.com

local ff1,ff2,k,x,d1,d2,d3,h

©Build function expressions
ff&"(xx+h)"→ff1
ff&"(xx-h)"→ff2

©Find first two estimates
.01→h
(expr(ff1)-expr(ff2))/.02→d1
.001→h
(expr(ff1)-expr(ff2))/.002→d2

©Loop to find best estimate
for k,4,14
 10^⁻k→h
 (expr(ff1)-expr(ff2))/(2*h)→d3
 if abs(d3-d2)>abs(d2-d1) then
  return d2
 else
  d2→d1
  d3→d2
 endif
endfor

©Best not found; return last estimate
return d3

Endfunc
```

Call *nder2()* with the function name as a string. For example, to find the derivative of tan(x) atπ /2.01, the call is nder2("tan",$\pi$/2.01)

which returns 16374.242. The absolute error is about 1.01E-4, and the relative error is 6.19E-9.

The table below demonstrates the improvement in the derivative estimate as *h* decreases, for tan(x) evaluated at x = 1.

| h | f'(x) | difference in f'(x) |
|---|---|---|
| 1E-2 | 3.4264 6416 009 | (none) |
| 1E-3 | 3.4255 2827 135 | 9.359E-4 |
| 1E-4 | 3.4255 1891 5 | 9.356E-6 |
| 1E-5 | 3.4255 1882 | 9.5E-8 |
| 1E-6 | 3.4255 188 | 2E-8 |
| 1E-7 | 3.4255 185 | 3E-7 |

*nder2()* starts with h = 0.01, and divides h by 10 for each new estimate, so that the steps for h are 1E-2, 1E-3, ... 1E-14. Since the difference in f'(x) starts increasing at h = 1E-7, *nder2()* returns the value for h = 1E-6.

*More accurate results with Ridders' method: nder1()*

Ridders' method *(Advances in Engineering Software*, vol. 4, no. 2, 1982*)* is based on extrapolating the central difference formula to h=0. This method also has the important advantage that it returns an estimate of the error, as well. This program, *nder2(),* implements the algorithm:

```
nder1(ff,xx,hh)
func
©("f",x,"auto" or h), return {f'(x),err}
©Based on Ridders' algorithm
©6jun00/dburkett@infinet.com

local con,con2,big,ntab,safe,i,j,err,errt,fac,amat,dest,fphh,fmhh,ffun,h1,d3

© Initialize constants
1.4→con
con*con→con2
1ᴇ900→big
10→ntab
2→safe
newmat(ntab,ntab)→amat

© Build function strings
ff&"(xx+hh)"→fphh
ff&"(xx-hh)"→fmhh
ff&"(xx)"→ffun

© Find starting hh if hh="auto"
if hh="auto" then
 if xx=0 then: .01→h1
 else: xx/1000→h1
 endif
 expr(ff&"(xx+h1)")-2*expr(ffun)+expr(ff&"(xx-h1)")→d3

 if d3=0: .01→d3
 (√(abs(expr(ffun)/(d3/(h1^2)))))/1Ø→hh
 if hh=Ø: .Ø1→hh

endif

©Initialize for solution loop
(expr(fphh)-expr(fmhh))/(2*hh)→amat[1,1]
big→err

©Loop to estimate derivative
for i,2,ntab
 hh/con→hh
 (expr(fphh)-expr(fmhh))/(2*hh)→amat[1,i]

 con2→fac

 for j,2,i
  (amat[j-1,i]*fac-amat[j-1,i-1])/(fac-1)→amat[j,i]
  con2*fac→fac
  max(abs(amat[j,i]-amat[j-1,i]),abs(amat[j,i]-amat[j-1,i-1]))→errt
  if errt≤err then
   errt→err
   amat[j,i]→dest
  endif

 endfor
 if abs(amat[i,i]-amat[i-1,i-1])≥safe*err:exit

endfor

return {dest,err}

endfunc
```

*nder1()* is called with the function name as a string, the evaluation point, and the initial step size. If the step size is "auto", then *nder1()* tries to find a good step size based on the function and evaluation point. *nder1()* returns a list with two elements. The first element is the derivative estimate, and the second element is the error estimate.

*nder1()* is called as

```
nder1(fname,x,h)
```

where *fname* is the name of the function as a string in double quotes. *x* is the point at which to find the derivative. *h* is a parameter which specifies a starting interval. If *h* is "auto" instead of a number, then *nder1()* will try to automatically select a good value for *h*.

For example, to find the derivative of tan(x) at x = 1.0, with an automatic step size, use

```
nder1("tan",1,"auto")
```

which returns {3.42551882077, 3.7E-11}. The derivative estimate is 3.4255..., and the error estimate is 3.7E-11. To find the same derivative with a manual step size of 0.1, use

```
nder1("tan",1,.1)
```

which returns {3.42551882081,1.4E-12}.

The value *h* is an initial step size. It should not be too small, in fact, it should be an interval over which the function changes substantially. I discuss this more in the section below, *More comments on nder1() and Ridders' method.*

To use *nder1()* and return just the derivative estimate, use

```
nder1(f,x,h)[1]
```

where *[1]* specifies the first element of the returned list.

If *nder1()* returns a very large error, then the starting interval *h* is probably the wrong value. For example,

`nder1("tan",1.57Ø7,"auto")` returns {-1.038873452988 E7, 3.42735731968 E8}

Note that the error is quite large, on the order of 3.43E8. We can manually set the starting interval to see if we can get a better estimate. First, using *nder1p()* (see below), we find that the starting interval with the "auto" setting is about 1.108E-4. If we try *nder1()* with a starting interval of about 1/10 of this value, we get

```
nder1("tan",1.57Ø7,1E-5) = {1.Ø7771965667E8, 1.62341}
```

Since the error estimate is much smaller, we can trust the derivative estimate.

Execution time will depend on the execution time of the function for which the derivative is being found. For simple functions, execution times of 5-20 seconds are not uncommon.

It can be convenient to have a user interface for *nder1().* This program provides an interface:

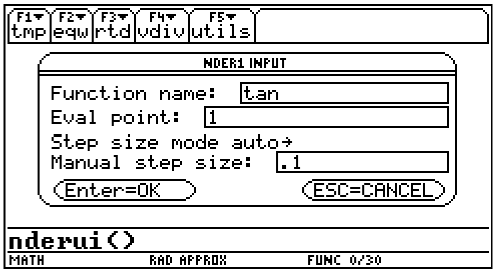
```
nderui()
Prgm
© User interface for nder1()
© 3janØØ/dburkett@infinet.com
© Result also saved in nderout
local fn1,xx,steps,smode,reslist,ssz

1→xx
.1→steps

lbl lp

string(xx)→xx
string(steps)→steps
dialog
title "NDER1 INPUT"
request "Function name",fn1
request "Eval point",xx
dropdown "Step size mode",{"auto","manual"},smode
request "Manual step size",steps
enddlog
if ok=Ø:return

expr(xx)→xx
expr(steps)→steps

when(smode=2,steps,"auto")→ssz

nder\nder1(fn1,xx,ssz)→reslist
```

```
reslist[1]→nderout

dialog
 title "NDER1 RESULTS"
 text "Input x: "&string(xx)
 text "dy/dx: "&string(reslist[1])
 text "Error est: "&string(reslist[2])
 text "(Derivative → nderout)"
enddlog
if ok=Ø:return
goto lp

EndPrgm
```

To use *nderui(), nder1()* must be saved in a folder called *nder*. *nderui()* should be stored in the same folder. Run *nderui()* like this:

```
nder/nderui()
```

and this input dialog box is shown:



NDER1 INPUT

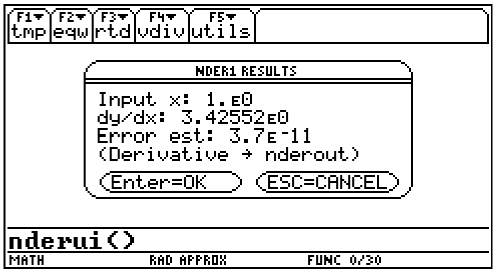
Function name: tan
Eval point: 1
Step size mode auto→
Manual step size: .1

nderui()

The Function Name is entered without quotes, as shown. *Eval point* is the point at which the derivative is found. *Step size mode* is a drop-down menu with these choices:

| | |
|---|---|
| 1: auto | *nder1() finds the interval size* |
| 2: manual | *you specify the interval in Manual step size* |

When all the input boxes are complete, press [ENTER] to find the derivative. This results screen is shown:



NDER1 RESULTS

Input x: 1.ᴇ0
dy/dx: 3.42552ᴇ0
Error est: 3.7ᴇ-11
(Derivative → nderout)

nderui()

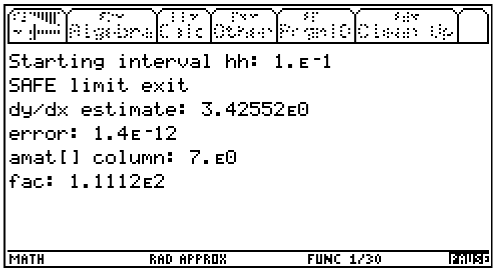
*Input x* is the point at which the derivative is estimated. *dy/dx* is the derivative estimate. *Error est* is the error estimate. The derivative estimate is saved in the global variable *nderout* in the current folder. Push [ENTER] to find another derivative, or press [ESC] to exit the program.

*Diagnosing nder1() with nder1p()*

*nder1p()* uses the same algorithm as *nder1(),* but is coded as a program instead of a function. This version can be used to debug situations in which *nder1()* returns results with unacceptably high errors.

*nder1p()* is called in the same way as *nder1():* `nder1p(f,x,h)`.

When *nder1p()* finishes, the results are shown on the program I/O screen. Push ENTER to return to the home screen. The results screen looks like this:



```
Starting interval hh: 1.ᴇ-1
SAFE limit exit
dy/dx estimate: 3.42552ᴇ0
error: 1.4ᴇ-12
amat[] column: 7.ᴇ0
fac: 1.1112ᴇ2
```

The 'Starting interval hh' shows the first value for *h*, which is especially helpful when the "auto" option is used. The string "SAFE limit exit" may or may not be shown, depending on how *nder1p()* terminates. The 'amat[] column' shows the number of main loops that executed before exit. 'fac' is a scaling factor.

*nder1p()* creates these global variables, which can all be deleted:

con, con2, big, ntab, safe, i, j, err, errt, fac, amat, dest, fphh, fmhh, ffun ,h1, d3

This is the code for *nder1p():*

```
nder1p(ff,xx,hh)
prgm
©("f",x,h) return f'(x), h is step size or "auto"
©6junØØ/dburkett@infinet.com
©program version of nder1()

1.4→con
con*con→con2
1ᴇ9ØØ→big
1Ø→ntab
2→safe

ff&"(xx+hh)"→fphh
ff&"(xx-hh)"→fmhh
ff&"(xx)"→ffun

if hh="auto" then
 if xx=Ø then: .Ø1→h1
 else: xx/1ØØØ→h1
 endif
 expr(ff&"(xx+h1)")-2*expr(ffun)+expr(ff&"(xx-h1)")→d3
 if d3=Ø: .Ø1→d3

 (√(abs(expr(ffun)/(d3/(h1^2)))))/1Ø→hh

 if hh=Ø: .Ø1→hh

endif

clrio
disp "Starting interval hh: "&string(hh)

newmat(ntab,ntab)→amat
(expr(fphh)-expr(fmhh))/(2*hh)→amat[1,1]
big→err

for i,2,ntab
 hh/con→hh
 (expr(fphh)-expr(fmhh))/(2*hh)→amat[1,i]

 con2→fac

 for j,2,i
  (amat[j-1,i]*fac-amat[j-1,i-1])/(fac-1)→amat[j,i]
  con2*fac→fac
  max(abs(amat[j,i]-amat[j-1,i]),abs(amat[j,i]-amat[j-1,i-1]))→errt
  if errt≤err then
   errt→err
   amat[j,i]→dest
  endif

 endfor
 if abs(amat[i,i]-amat[i-1,i-1])≥safe*err then
 disp "SAFE limit exit"
 exit
 endif

endfor

disp "dy/dx estimate: "&string(dest)
disp "error: "&string(err)
disp "amat[] column: "&string(i)
disp "fac: "&string(fac)
pause
disphome

endprgm
```

*General comments on numerical differentiation*

Any numerical derivative routine is going to suffer accuracy problems where the function is changing rapidly. This includes asymptotes. For example, suppose that we try to find the derivative of tan(1.5707). This is very close to the asymptote of pi/2 = 1.57079632679. *nder2()* returns an answer of -1.009...E6, which is clearly wrong. The problem is that the starting interval brackets the asypmtote. *nder1()* returns an answer of -1.038..E7, but at least also returns an error of 3.427E8, so we know something is wrong.

The function must be continuous on the sample interval for both *nder1()* and *nder2().*

There are other methods for finding numeric derivatives. For example, you can fit a polynomial (either Lagrange or least-squares) through some sample points, then find the derivative of the polynomial. These methods might not have any speed advantage over *nder2(),* because an accurate, high-order polynomial requires considerable time to find the coefficients, although it is fast to find the derivative once you have them.

In my library of numerical methods books I find little reference to finding numerical derivatives. This is perhaps because it is relatively easy to find derivatives of most simple functions.

*Numerical Recipes in Fortran*, 2nd edition, William H. Press et al. Section 5.7, p181 describes various issues and problems in calculating numerical derivatives. The expressions for the errors of *avgRC()* and *nDeriv()* are also found here.

*More comments on nder1() and Ridders' method*

*nder1()* is a 'last resort' for finding numerical derivatives, to be used when the other alternatives have failed. The other alternatives are 1) finding a symbolic derivative, and 2) using the 89/92+ built-in numerical derivative functions *avgRC()* and *nDeriv().*

For example, there is no point in using *nder1()* for a simple function such as tan(x). The 89/92+ can easily find the symbolic derivative, which can then be evaluated numerically. However, you may have complex, programmed functions for which 89/92+ cannot find a symbolic derivative.

The error will increase dramatically near function asymptotes, where the function is changing very rapidly.

It is possible to adjust *h* in *nDeriv()* to get reduce the error at a given point, but that is not very valuable if you don't know what the answer should be! However, it is possible with a program like *nDer1(),* which returns an error estimate, to improve the answer by adjusting *h*. For example, suppose we want to find the derivative of tan(x) as accurately as possible at x = 1.4. The basic idea is to call *nder()* with different values of *h*, and try to locate the value of *h* with the smallest error. The table below shows some results.

| **Interval size *h*** | **Reported error** | **Actual error** |
|---|---|---|
| 0.13 | 1.77E-9 | 1.06E-9 |
| 0.10 | 7.54E-10 | 5.78E-10 |
| 0.08 | 1.03E-9 | 2.57E-10 |
| 0.06 | 4.58E-10 | 4.28E-10 |
| 0.04 | 4.06E-10 | 2.71E-10 |
| 0.02 | 3.83E-10 | 1.44E-9 |
| 0.01 | 9.06E-10 | 6.66E-10 |
| 0.005 | 3.05E-9 | 6.97E-9 |

This example shows that the error is not too sensitive to the interval size, and the actual error is usually less than the reported error bound.

*nder1()* uses Ridders' algorithm to estimate the derivative. The general idea is to extrapolate the central-difference equation to h=0:

$$f'(x) = \frac{f(x+h)-f(x-h)}{2\cdot h}$$

If we could let h=0, then this equation would return the exact derivative. However, h=0 is obviously not allowed. Further, we can't just make *h* arbitrarily small, because the loss of precision degrades the answer. This can be seen by using *nDeriv()* with various values of h. The table below shows the results returned by *nDeriv()* for f(x) = tan(x), with various values of h, and with x = 1

| h | f'(x) | error |
|---|---|---|
| 1E-01 | 3.5230 0719 849 | -9.749E-02 |
| 1E-02 | 3.4264 6416 008 | -9.453E-04 |
| 1E-03 | 3.4255 2827 133 | -9.451E-06 |
| 1E-04 | 3.4255 1891 538 | -9.456E-08 |
| 1E-05 | 3.4255 1882 37 | -2.886E-09 |
| 1E-06 | 3.4255 1880 37 | 1.712E-08 |
| 1E-07 | 3.4255 1882 081 | -1.000E-13 |
| 1E-08 | 3.4255 1882 081 | -1.000E-13 |
| 1E-09 | 3.4255 1882 081 | -1.000E-13 |
| 1E-10 | 3.4255 1882 081 | -1.000E-13 |
| 1E-11 | 3.4272 3158 023 | -1.000E-13 |
| 1E-13 | 3.5967 9476 186 | -1.713E-13 |
| 1E-14 | 1.7127 5941 041 | 1.712E+00 |

This seems to be very good performance - too good, in fact. Suppose that instead of including the tan(x) function in *nDeriv(),* we call the function indirectly, like this:

```
nDeriv(ftan(xx),xx,h)|xx=1
```

where *ftan()* is just a user function defined to return tan(x). This results in the following:

| h | f'(x) | error |
|---|---|---|
| 1E-01 | 3.5230071984915 | -9.75E-02 |
| 1E-02 | 3.426464160085 | -9.45E-04 |
| 1E-03 | 3.42552827135 | -9.45E-06 |
| 1E-04 | 3.425518915 | -9.42E-08 |
| 1E-05 | 3.42551882 | 8.15E-08 |
| 1E-06 | 3.4255188 | 2.08E-08 |
| 1E-07 | 3.4255185 | 3.21E-07 |
| 1E-08 | 3.42552 | -1.18E-06 |
| 1E-09 | 3.4255 | -1.88E-05 |
| 1E-10 | 3.4255 | -1.88E-05 |
| 1E-11 | 3.425 | 5.19E-04 |
| 1E-12 | 3.4 | 2.55E-02 |
| 1E-13 | 3.5 | -7.45E-02 |
| 1E-14 | 0 | 3.43E+00 |

This is more typical of the performance we would expect from the central difference formula. As *h* decreases, the number of digits in the results decreases, because of the increasingly limited resolution of f(x+h) - f(x-h). The accuracy gets better until h = 1E-6, then starts getting worse. At the best error, we only have 8 significant digits in the result.

This example seems to imply that the 89/92+ does not directly calculate the central difference formula to estimate the derivative of tan(x). Instead, the 89/92+ uses trigonometric identities to convert

$$\frac{\tan(x+h)-\tan(x-h)}{2\cdot h}$$

to

$$\frac{\sin(x+h)(\cos(x-h)-\cos(x+h)\sin(x-h))}{2\cdot h\cdot\cos(x+h)\cos(x-h)}$$

This is a clever trick, because it converts the tangent function, which has asymptotes, into a function of sines and cosines, which don't have asymptotes. Any numerical differentiator is going to have trouble wherever the function is changing rapidly, such as near asymptotes.

*nDeriv()* also transforms some other built-in functions:

| | |
|---|---|
| sinh(), cosh(), tanh() | uses the exponential definitions of the hyperbolic functions |
| log() | converts expression to natural log |
| $e^x$ | converts expression to use e^x form of sinh() |
| $10^x$ | converts expression to use sinh() |
| $\tanh^{-1}()$ | converts expression to use ln() |

but *nDeriv()* directly evaluates these functions:

ln(), $\sin^{-1}()$, $\cos^{-1}()$, $\tan^{-1}()$, $\sinh^{-1}()$, $\cosh^{-1}()$

*nDeriv()* also simplifies polynomials before calculating the central difference formula. For example, *nDeriv()* converts

$3x^2 + 2x + 1$

to

$6(x + 1/3) = 6x + 2$

Notice that in this case *h* drops out completely, because the central difference formula calculates derivatives of 2nd-order equations exactly.

While this is a laudable approach in terms of giving an accurate result, it is misleading if you expect *nDeriv()* to really return the actual result of the central difference formula. Further, you can't take advantage of this method with your own functions if those functions are not differentiable by the 89/92+. This example establishes the need for an improved numerical differentiation routine.

In Ridders' method the general principle is to extrapolate for h=0. *nder1()* implements this idea by using successively smaller values of the starting interval *h*. At each value of *h*, a new central difference estimate is calculated. This new estimate, along with the previous estimates, is used to find higher order estimates. In general, the error will get better as the starting value of *h* is increased, then suddenly get very large. The table below shows this effect using *nder1()* for f(x) = tan(x), where x=1.

| hh | error |
|---|---|
| 0.001 | 3.74E-10 |
| 0.005 | 1.48E-10 |
| 0.01 | 2.22E-11 |
| 0.1 | 4.70E-12 |
| 0.15 | 2.00E-12 |
| 0.2 | 1.64E-11 |
| 0.3 | -1.10E-12 |
| 0.4 | -4.60E-12 |
| 0.5 | 4.93E-01 |

Note that the error suddenly increases at hh = 0.5. Obviously, there is a best value for *hh* that reduces the error, but this best value is not too sensitive to the actual value of hh, as the error is on the order of E-12 from *hh* = 0.1 to 0.4.

One way to find the best *hh* would be to try different values of *hh* and see which returned the smallest error estimate. Since *nder1()* is so slow, I wanted a better method. In the reference below, the authors suggest that a value given by

$$h = \left[ \frac{f(x)}{f''(x)} \right]^{\frac{1}{2}}$$

minimizes the error. However, note that this expression includes the second derivative of the function, f''(x). While we don't know this (we don't even know the first derivative!), we can estimate it with an expansion of the central difference formula, modified to find the second derivative instead of the first:

$$f''(x) \simeq \frac{d3}{h_1^2}$$

where $h_1$ is a small interval, and

$$d3 = f(x+h_1) -2f(x) + f(x-h_1)$$

It might seem that we have just exchanged finding one interval, *hh*, for another interval, $h_1$. However, since we are just trying to find a crude estimate to f''(x), it turns out that we don't have to have a precise value for $h_1$. I arbitrarily chose

$h_1 = x/1000$ if $x \neq 0$, or
$h_1 = 0.1$ if $x=0$

If the function is fairly linear near *x*, d3 may equal 0. If this happens, it means that f'(x) is also near zero, so we can use any small value for d3; I chose d3 = 0.01, which avoids division by zero in the equation for hh above.

Next, if f(x) = 0, we'll get h = 0, which won't work. In this case, I set h = 0.01.

So, the final equation is

$$h = \frac{\sqrt{\frac{abs(f(x))}{f''(x)}}}{10}$$

where I have used the absolute value *abs()* to ensure that the root is real. I also divide the final result by 10 to ensure that *h* is small enough that *nder1()* doesn't terminate too early.

*nder1()* can terminate in one of two ways: if the error keeps decreasing, *nder1()* will run until the *amat[]* matrix is filled with values. However, if at some step the error increases, *nder1()* will terminate early. The variable *safe* specifies the magnitude of the error increase that causes termination. Refer to the code listing for details.

From my testing with a few functions, *nder1()* nearly always terminates by exceeding the *safe* limit.

### [6.27] Find Bernoulli numbers and polynomials

*[Note: since this tip was written, Bhuvanesh Bhatt has also written a function to find both Bernoulli numbers and polynomials. Bhuvanesh' function is smaller and handles complex arguments. You can get it at his site (see the "More Resources - Web sites" section for the URL), and at ticalc.org. Also, M. Daveluy has written a Bernoulli number function, which can be found at ti-cas.org.]*

Bernoulli numbers are generated from the Benoulli polynomials evaluated at zero. Bernoulli polynomials are defined by the generating function

$$\frac{te^{xt}}{e^{t}-1} = \sum_{n=0}^{\infty} B_n(x)\frac{t^n}{n!}$$

Bernoulli polynomials can also be defined recursively by

$$B_0(x) = 1 \qquad [1]$$

$$\frac{d}{dx}B_n(x) = nB_{n-1}(x) \qquad [2]$$

$$\int_0^1 B_n(x)dx = 0 \quad \text{for } n \geq 1 \qquad [3]$$

The first few Bernoulli polynomials are

$$B_0(x) = 1 \qquad B_3(x) = \frac{2x^3-3x^2+x}{2}$$

$$B_1(x) = \frac{2x-1}{2} \qquad B_4(x) = \frac{30x^4-60x^3+30x^2-1}{30}$$

$$B_2(x) = \frac{6x2-6x+1}{6} \qquad B_5(x) = \frac{6x^5-15x^4+10x^3-x}{6}$$

The nth Bernoulli number is denoted as $B_n$. The Bernoulli numbers can be defined by the generating function

$$\frac{t}{e^{t}-1} = \sum_{n=0}^{\infty} B_n\frac{t^n}{n!}$$

or, as mentioned above, by evaluating $B_n(0)$. However, a faster method to find Bernoulli numbers on the 89/92+ uses this identity:

$$\xi(2n) = \frac{(-1)^{n-1} B_{2n} (2\pi)^{2n}}{2(2n)!}$$

The notation *2n* is used since this identity is only true for even integers. $\xi(n)$ is the Riemann Zeta function,

$$\xi(n) = \sum_{k=1}^{\infty} \frac{1}{k^n}$$

Bernoulli numbers for odd *n* > 1 are zero. The first few non-zero Bernoulli numbers are

$B_0 = 1$ $B_1 = -1/2$ $B_2 = 1/6$ $B_4 = -1/30$ $B_6 = 1/42$

It turns out that the 89/92+ can evaluate $\xi(n)$ very quickly for even *n*, which is what we need. Solving the identity above for $B_{2n}$ gives

$$B_{2n} = \frac{2\xi(2n)\cdot(2n)!}{(-1)^{n-1}(2\pi)^{2n}}$$

This function returns the Bernoulli number $B_n$:

```
bn(n)
Func
©Bernoulli number Bn
©21junØØ/dburkett@infinet.com

if n=Ø:return 1
if n=1:return ⁻1/2
if n<Ø:return undef
if fpart(n/2)≠Ø:return Ø

(Σ(z^⁻n,z,1,∞)*2*n!)/(((⁻1)^(n/2-1))*(2π)^n)

EndFunc
```

The first three executable lines handle the special cases for $B_0 = 1$, $B_1 = -1/2$ and $B_n$ undefined when n<0. The fourth line returns zero for odd *n* where n>1. Note that the expression to find $B_n$ has been transformed from an expression in *2n* to an expression in *n*.

Finding the Bernoulli polynomials is a little more complicated, but can still be done on the 89/92+. The program uses the recursive definition given above in equations [1], [2] and [3]. First, take the antiderivative of equation [2] to find

$$B_n(x) = \int nB_{n-1}(x)dx$$

Since these integrals are simple polynomials, the 89/92+ can easily find the symbolic integral. I use the definite integral of equation [3] to find the constant of integration:

$$ff1 = \int nB_n(x)dx$$

$$ff2 = ff1 - \int_0^1 ff1(x)dx$$

To find the *n*th Bernoulli polynomial requires finding all the (n-1) previous polynomials, so this is time-consuming for higher-order polynomials. For this reason I wrote two different versions of programs to find the Bernoulli polynomials. One version is a function which returns a single polynomial of the order specified by the function argument. The second version builds a table of the polynomials up to a specified order. Individual polynomials can be quickly retrieved from this table by another function.

bnpolys(n) returns a single Bernoulli polynomial of order *n* as a function of *x:*

```
bnpolys(n)
Func
©(n) return Bernoulli polynomial of order n
©27jun00/dburkett@infinet.com

local k,f,g

x-1/2→f
if n=1:return f

for k,2,n
 ∫(k*f,x)→g
 g-∫(g,x,0,1)→f
endfor

return f

EndFunc
```

*bnpolys()* may not return the polynomial in the form you want. For example, *bnpolys(3)* returns

$$\frac{x\left(2x^2-3x+1\right)}{2}$$

Use *expand()* to convert this result to $x^3 - \frac{3x^2}{2} + \frac{x}{2}$

or use *comdenom()* to get $\frac{2x^3-3x^2+x}{2}$

*bnpolys()* slows down for large arguments. For example, *bnpolys(20)* takes about 22 seconds on my HW2 92+ with AMS 2.04, and *bnpolys(50)* takes about 160 seconds. This long execution time occurs because *bnpolys()* must calculate all the polynomials of order n-1, to return the polynomial of order *n.* If you frequently use higher-order polynomials, it is worthwhile to build a table in advance, then use a simple routine that just recalls the appropriate polynomial.

These routines perform those functions. *bnpoly()* builds the list of polynomials, and *bpo()* is a function that returns a given polynomial, after *bnpoly()* is run.

First, use *bnpoly()* to build the list of polynomials:

```
bnpoly(nxx)
prgm
©(n) Fill Bernoulli polynomial list bpoly[] up to n.
```

```
©1julØØ/dburkett@infinet.com
©Save polynomials in list bpoly[].

local k,k1,k2,bpdim,ff1,usermode,choice

©Save user's mode; set modes
getmode("all")→usermode
setmode({"Complex Format","Real","Vector
Format","Rectangular","Exact/Approx","Exact","Pretty Print","Off"})

©If bpoly[] exists unarchive it, else create it & initialize it.
if gettype(spfn\bpoly)="NONE" then
 newlist(1)→spfn\bpoly
 x-1/2→spfn\bpoly[1]
else
 unarchiv(spfn\bpoly)
endif

dim(spfn\bpoly)→bpdim
clrio

©Loop to derive Bernoulli polynomials
if nxx>bpdim then
 augment(spfn\bpoly,newlist(nxx-bpdim))→spfn\bpoly
 for k,bpdim+1,nxx
  ∫(k*spfn\bpoly[k-1],x)→ff1
  ff1-∫(ff1,x,Ø,1)→spfn\bpoly[k]
  disp k
  disp spfn\bpoly[k]
 endfor
endif

©Prompt to archive bpoly[]
1→choice
dialog
 title "BNPOLY"
 dropdown "Archive polynomial list?",{"yes","no"},choice
enddlog

if ok=1 and choice=1 then
 archive spfn\bpoly
endif

©Restore user's modes
setmode(usermode)
clrio
disphome

Endprgm
```

Then, use *bpo()* to recall a particular polynomial:

```
bpo(kk)
func
©(k) Bernoulli polynomial Bk
©1julØØ/dburkett@infinet.com
©Generates "Domain error" if kk>dim(bpoly)

if kk<Ø:return "bpo arg error"

when(kk=Ø,1,spfn\bpoly[kk])
```

```
    Endfunc
```

These routines must both be installed in a folder called *\spfn*. The list of Bernoulli polynomials will be created in this same folder. To create a list of polynomials, execute

```
    bnpoly(n)
```

where *n* is the highest-order polynomial you expect to use. For example, if you use polynomials up to the 20th order, then use *bnpoly(20).* If you later need higher-order polynomials, just run *bnpoly()* again, and it will append the additional polynomials to the list.

As *bnpoly()* creates the polynomials, it displays them in the program I/O screen. After all the polynomials are created, a dialog box is shown with the prompt

```
    Archive polynomial list?
```

Answer 'yes' to archive the list in flash memory, or 'no' to leave it in RAM. I provide this option because the list is large for high order polynomials, and there is no advantage to having it in RAM. If you later increase the size of the list, *bnpoly()* will unarchive it for you.

This table gives you some idea of the memory required for tables of various orders of polynomial. The size is in bytes with the list archived. The size of the last polynomial in the list is also shown

**Size of bpoly[] and last polynomial**

| Polynomial order | Total list size (bytes) | Last polynomial size (bytes |
|---|---|---|
| 10 | 400 | 71 |
| 20 | 1,463 | 155 |
| 30 | 3,370 | 264 |
| 50 | 10,525 | 527 |
| 75 | 28,581 | 967 |
| 100 | 61,327 | 1,609 |

Since the maximum size of an 89/92+ variable is about 64K, the maximum order that can be saved is about 100.

Operation of *bpo()* is simple: just call *bpo(k),* where *k* is  the the order of the polynomial to return. For example, if *bnpoly()* was executed to create a list of polynomials up to order 30, then to return the 10-th order polynomial, use

```
    bpo(30)
```

*bpo()* does two things, beyond simply returning the polynomial in *bpoly[k].* First, it returns an error message if k<0. Second, it returns 1 for k=0, since $B_0$(x) is 1, and $B_0$(x) is not stored in *bpoly[].*

I would usually include a test to ensure that *k* is less than the list size, like this:

```
    if kk<0 or kk>dim(spfn\bpoly): return "bpo arg error"
```

However, it turns out that the 89/92+ are *extremely* slow to find the dimension of lists with large elements, even if the lists have few elements. For example, if the bpoly[] has 70 polynomials, its size is about 24000 bytes, and `dim(spfn\bpoly)` takes over 30 seconds to return the dimension of 70! If bpoly[] has 100 elements, `dim(bpoly)` takes 5 seconds to fail with a "Memory" error message.

This explains the long delay when *bnpoly()* is used to add polynomials to lists that are already of high order.

One potential work-around to this problem would be to use the `try...else...endtry` condtional test to trap the error, but *functions* cannot use `try...endtry`!

So, rather than do a proper test for a valid input argument, I accept that fact that *bpo()* will not fail gracefully, in return for fast execution times for proper arguments.

For more information on Bernoulli numbers and polynomials, these references may be helpful:

*Handbook of Mathematical Functions*, Milton Abramowitz and Irene A. Stegun, Dover, 1965. This reference defines the Bernoulli numbers and polynomials, has a very complete table of properties, and also tables of the polynomials and numbers. Various applications are shown throughout the book where relevant.

*Numerical Methods for Scientists and Engineers,* R.W. Hamming, Dover, 1962. Hamming shows another method to generate the Bernoulli numbers using a summation (p190).

*Ada and the first computer*, Eugene Eric Kim and Betty Alexandra Toole, Scientific American magazine, May 1999. This very interesting article describes a program written in 1843, by Ada, countess of Lovelace, for Charles Babbage's Analytical Engine. The purpose of the program was to calculate the Bernoulli numbers. The analytical engine, a mechanical computer, was never completed, as the British government didn't fund it adequately, and Babbage kept revising the design.

These web sites are also interesting:

*http://www.treasure-troves.com/math/BernoulliNumber.html*
This site describes the derivation and basic properties of the Bernoulli numbers and polynomials.

*http://venus.mathsoft.com/asolve/constant/apery/brnlli.html*
This site shows the expression for the tangent function, as a function of the Bernoulli number.

*http://www-history.mcs.st-andrews.ac.uk/~history/Mathematicians/Bernoulli_Jacob.html*
This site has a very nice biography of Jacob Bernoulli, who was the particular Bernoulli responsible for the Bernoulli numbers and polynomials.

### [6.28] Bilinear interpolation

Tip [6.22] shows how to do linear interpolation in two dimensions, that is, to estimate y = f(x), given two points. This basic concept can be extended to three dimensions, in that we want to estimate z = f(x,y). In this tip, I show two methods two interpolate in three dimensions. The first method, called bilinear interpolation, requires four (x,y,z) points and performs a linear interpolation. The second method, which I call 9-point interpolation, requires nine (x,y,z) points, but fits the data to a general second-order polynomial in *x* and *y.* This method is slower, but more accurate.

### *Bilinear interpolation*

If we are given four (x,y) points such that

$$\begin{array}{lll} f(x_1,y_1) = z_1 & \text{and} & x_2 > x_1 \\ f(x_1,y_2) = z_2 & & y_2 > y_1 \\ f(x_2,y_1) = z_3 & & \\ f(x_2,y_2) = z_4 & & \end{array}$$

then we can solve for a function in four unknown coefficients. There are an infinite number of such equations, but the most simple candidate is

$$z = ax + bx + cxy + d$$

so the system to solve for *a*, *b*, *c* and *d* is

$$\begin{array}{l} ax_1 + by_1 + cx_1y_1 + d = z_1 \\ ax_1 + by_2 + cx_1y_2 + d = z_2 \\ ax_2 + by_1 + cx_2y_1 + d = z_3 \\ ax_2 + by_2 + cx_2y_2 + d = z_4 \end{array}$$

We could solve this directly for *a*, *b*, *c* and *d*, but we can get a more simple solution by scaling x and y, like this:

$$t = \frac{x-x_1}{x_2-x_1} \qquad \text{and} \qquad u = \frac{y-y_1}{y_2-y_1}$$

With this scaling, t = 0 when $x = x_1$ and t = 1 when $x = x_2$. Similarly, u = 0 when $y = y_1$ and u = 1 when $y = y_2$. This equation system to solve simplifies to this:

$$\begin{array}{lll} d = z_1 & \text{or} & a = z_3 - z_1 \\ b + d = z_2 & & b = z_2 - z_1 \\ a + d = z_3 & & c = z_1 - z_2 - z_3 + z_4 \\ a + b + c + d = z_4 & & d = z_1 \end{array}$$

So the equation to estimate *z* in terms of *t* and *u* is

$$z = (z_3 - z_1)t + (z_2 - z_1)u + (z_1 - z_2 - z_3 + z_4)ut + z_1$$

We can expand this equation, collect on $z_1$, $z_2$, $z_3$ and $z_4$, and further factor that result to get

$$z = z_1(1-u)(1-t) + z_2u(1-t) + z_3t(1-u) + z_4ut$$

The function *bilinint()* does the interpolation with this formula

```
bilinint(xa,xb,ya,yb,zaa,zab,zba,zbb,x,y)
Func
©(xa,xb,ya,yb,zaa,zab,zba,zbb,x,y) Lin interpolate z=f(x,y)
©8octØØ dburkett@infinet.com

local t,u

(x-xa)/(xb-xa)→t
```

```
(y-ya)/(yb-ya)→u

(1-t)*(1-u)*zaa+t*(1-u)*zba+t*u*zbb+(1-t)*u*zab

EndFunc
```

Note that the variable names have been changed to avoid conflict with built-in variables. The equivalence is

| | | | |
|---|---|---|---|
| xa == $x_1$ | ya == $y_1$ | zaa == $z_1$ | zab == $z_2$ |
| xb == $x_2$ | yb == $y_2$ | zba == $z_3$ | zbb == $z_4$ |

The z-variables are named so it is easy to remember which z-variable goes with a particular (x,y) data point. The second two characters of each z-variable indicate the corresponding x- and y-variables:

zaa = f(xa,ya)   zab = f(xa,yb)   zba = f(xb,ya)   zbb = f(xb,yb)

As an example, given this data:

| | ya = 0.2 | yb = 0.3 |
|---|---|---|
| xa = 0.5 | zaa = 0.4699 | zab = 0.4580 |
| xb = 0.6 | zba = 0.5534 | zbb = 0.5394 |

to interpolate for x = 0.52 and y = 0.28, the call is

```
bilinint(.5,.6,.2,.3,.4699,.458Ø,.5534,.5394,.58,.28)
```

which returns 0.4767.

This program, *bilinui(),* provides a user interface for the interpolation function.

```
bilinui()
Prgm
©User interface for bilinint()
©8octØØ dburkett@infinet.com

local z

if gettype(x1)="NONE" then
 Ø→xa:2→xb:Ø→ya:2→yb
 Ø→zaa:1→zab:2→zba:3→zbb
 1→x:1→y
endif

lbl l1

string(xa)→xa:string(xb)→xb
string(ya)→ya:string(yb)→yb
string(zaa)→zaa:string(zab)→zab
string(zba)→zba:string(zbb)→zbb

dialog
 title "bilinui"
 request "x1",xa
 request "x2",xb
 request "y1",ya
 request "y2",yb
 request "f(x1,y1)",zaa
 request "f(x1,y2)",zab
 request "f(x2,y1)",zba
 request "f(x2,y2)",zbb
enddlog

expr(xa)→xa
expr(xb)→xb
expr(ya)→ya
expr(yb)→yb
expr(zaa)→zaa
expr(zab)→zab
expr(zba)→zba
expr(zbb)→zbb

if ok=0:return

string(x)→x
string(y)→y

dialog
 title "bilinui"
 request "x",x
 request "y",y
enddlog

expr(x)→x
expr(y)→y

if ok=0:return

bilinint(xa,xb,ya,yb,zaa,zab,zba,zbb,x,y)→z

dialog
 title "bilinui result"
 text "z= "&string(z)
enddlog

if ok=0:return

goto l1

EndPrgm
```

This program accepts the user input in dialog boxes, and displays the result in a dialog box. The previous inputs are saved in global variables, so they can be used as defaults the next time you run the program. These global variables are

xa, xb, ya, yb, zaa, zab, zba, zbb, x, y

You can press [ESC] at any dialog box to exit the program. Otherwise, the program continues to run, so that you can do more than one interpolation at once.

Note that the input must be done in two dialog boxes, since there are ten input elements, and a single dialog box can hold only eight. The program calls *bilinint()* to perform the actual interpolation. *bilinui()* and *bilinint()* must be in the same folder, and that folder must be the current folder.

This method is good enough for many problems, assuming that the function changes slowly, and the x- and y-values are close enough together such that the function is 'close' to linear in both variables. The interpolated values change smoothly and gradually within a particular x and y pair, but the derivatives change abruptly as you move to the next row or column of the original data table. This may or may not matter in your application. If it matters, there are more sophisticated interpolation methods which force a continuous derivative across row and column boundaries. For more discussion, refer to

*http://lib-www.lanl.gov/numerical/bookfpdf/f3-6.pdf*

which is the link to the appropriate section in the book *Numerical Recipes in Fortran*.

### *9-point interpolation*

Bilinear interpolation cannot account for any curvature in the function to be interpolated, since it is, by definition, linear. The 9-point interpolation method essentially performs a Lagrangian interpolating polynomial fit on the function

$$z = a \cdot x^2 \cdot y^2 + b \cdot x^2 \cdot y + c \cdot x \cdot y^2 + d \cdot x^2 + e \cdot y2 + f \cdot x \cdot y + g \cdot x + h \cdot y + i \qquad [1]$$

Since the interpolating polynomial includes squared terms of each independent variable, it can match some curvature in the underlying function.

We need to find the coefficients *a, b, c ...*, then we can calculate *z* for *x* and *y.* We have nine equations in nine unknowns if we use the points in the table function to interpolate like this

| | $y_1$ | $y_2$ | $y_3$ |
|---|---|---|---|
| $x_1$ | $z_1$ | $z_2$ | $z_3$ |
| $x_2$ | $z_4$ | $z_5$ | $z_6$ |
| $x_3$ | $z_7$ | $z_8$ | $z_9$ |

We choose $x_2$ and $y_2$ as the closest values in the table to the *x* and *y* at which we want to interpolate. This results in a linear system of nine equations:

$$z_1 = a \cdot x_1^2 \cdot y_1^2 + b \cdot x_1^2 \cdot y_1 + c \cdot x_1 \cdot y_1^2 + d \cdot x_1^2 + e \cdot y_1^2 + f \cdot x_1 \cdot y_1 + g \cdot x_1 + h \cdot y_1 + j \qquad [2]$$

$$z_2 = a \cdot x_1^2 \cdot y_2^2 + b \cdot x_1^2 \cdot y_2 + c \cdot x_1 \cdot y_2^2 + d \cdot x_1^2 + e \cdot y_2^2 + f \cdot x_1 \cdot y_2 + g \cdot x_1 + h \cdot y_2 + j$$

$$z_3 = a \cdot x_1^2 \cdot y_3^2 + b \cdot x_1^2 \cdot y_3 + c \cdot x_1 \cdot y_3^2 + d \cdot x_1^2 + e \cdot y_3^2 + f \cdot x_1 \cdot y_3 + g \cdot x_1 + h \cdot y_3 + j$$

$$z_4 = a \cdot x_2^2 \cdot y_1^2 + b \cdot x_2^2 \cdot y_1 + c \cdot x_2 \cdot y_1^2 + d \cdot x_2^2 + e \cdot y_1^2 + f \cdot x_2 \cdot y_1 + g \cdot x_2 + h \cdot y_1 + j$$

$$z_5 = a \cdot x_2^2 \cdot y_2^2 + b \cdot x_2^2 \cdot y_2 + c \cdot x_2 \cdot y_2^2 + d \cdot x_2^2 + e \cdot y_2^2 + f \cdot x_2 \cdot y_2 + g \cdot x_2 + h \cdot y_2 + j$$

$$z_6 = a \cdot x_2^2 \cdot y_3^2 + b \cdot x_2^2 \cdot y_3 + c \cdot x_2 \cdot y_3^2 + d \cdot x_2^2 + e \cdot y_3^2 + f \cdot x_2 \cdot y_3 + g \cdot x_2 + h \cdot y_3 + j$$

$$z_7 = a \cdot x_3^2 \cdot y_1^2 + b \cdot x_3^2 \cdot y_1 + c \cdot x_3 \cdot y_1^2 + d \cdot x_3^2 + e \cdot y_1^2 + f \cdot x_3 \cdot y_1 + g \cdot x_3 + h \cdot y_1 + j$$

$$z_8 = a \cdot x_3^2 \cdot y_2^2 + b \cdot x_3^2 \cdot y_2 + c \cdot x_3 \cdot y_2^2 + d \cdot x_3^2 + e \cdot y_2^2 + f \cdot x_3 \cdot y_2 + g \cdot x_3 + h \cdot y_2 + j$$

$$z_9 = a \cdot x_3^2 \cdot y_3^2 + b \cdot x_3^2 \cdot y_3 + c \cdot x_3 \cdot y_3^2 + d \cdot x_3^2 + e \cdot y_3^2 + f \cdot x_3 \cdot y_3 + g \cdot x_3 + h \cdot y_3 + j$$

In matrix form this system is written as

$$\begin{bmatrix} x_1^2 \cdot y_1^2 & x_1^2 \cdot y_1 & x_1 \cdot y_1^2 & x_1^2 & y_1^2 & x_1 y_1 & x_1 & y_1 & 1 \\ x_1^2 \cdot y_2^2 & x_1^2 \cdot y_2 & x_1 \cdot y_2^2 & x_1^2 & y_2^2 & x_1 y_2 & x_1 & y_2 & 1 \\ x_1^2 \cdot y_3^2 & x_1^2 \cdot y_3 & x_1 \cdot y_3^2 & x_1^2 & y_3^2 & x_1 y_3 & x_1 & y_3 & 1 \\ x_2^2 \cdot y_1^2 & x_2^2 \cdot y_1 & x_2 \cdot y_1^2 & x_2^2 & y_1^2 & x_2 y_1 & x_2 & y_1 & 1 \\ x_2^2 \cdot y_2^2 & x_2^2 \cdot y_2 & x_2 \cdot y_2^2 & x_2^2 & y_2^2 & x_2 y_2 & x_2 & y_2 & 1 \\ x_2^2 \cdot y_3^2 & x_2^2 \cdot y_3 & x_2 \cdot y_3^2 & x_2^2 & y_3^2 & x_2 y_3 & x_2 & y_3 & 1 \\ x_3^2 \cdot y_1^2 & x_3^2 \cdot y_1 & x_3 \cdot y_1^2 & x_3^2 & y_1^2 & x_3 y_1 & x_3 & y_1 & 1 \\ x_3^2 \cdot y_2^2 & x_3^2 \cdot y_2 & x_3 \cdot y_2^2 & x_3^2 & y_2^2 & x_3 y_2 & x_3 & y_2 & 1 \\ x_3^2 \cdot y_3^2 & x_3^2 \cdot y_3 & x_3 \cdot y_3^2 & x_3^2 & y_3^2 & x_3 y_3 & x_3 & y_3 & 1 \end{bmatrix} \cdot \begin{bmatrix} a \\ b \\ c \\ d \\ e \\ f \\ g \\ h \\ i \end{bmatrix} = \begin{bmatrix} z_1 \\ z_2 \\ z_3 \\ z_4 \\ z_5 \\ z_6 \\ z_7 \\ z_8 \\ z_9 \end{bmatrix} \qquad [3]$$

or, in terms of equivalent matrix variables

$$X \cdot a = z \qquad [4]$$

We could solve this system for the coefficients, but that requires inverting the *X* matrix. We can avoid inverting the matrix, and get the solution more quickly, if we scale the x- and y-coordinates such that

$$x_1 \equiv u_1 = 0 \qquad x_2 \equiv u_2 = 1 \qquad x_3 \equiv u_3 = 2 \qquad [5]$$

$$y_1 \equiv v_1 = 0 \qquad y_2 \equiv u_2 = 2 \qquad y_3 \equiv v_3 = 2 \qquad [6]$$

We scale the original *x* and *y* variables (at which to interpolate) with

$$u = \begin{Bmatrix} \frac{x - x_1}{x_2 - x_1} \mid x < x_2 \\ \frac{x - x_2}{x_3 - x_2} + 1 \mid x \geq x_2 \end{Bmatrix} \qquad \text{and} \qquad v = \begin{Bmatrix} \frac{y - y_1}{y_2 - y_1} \mid y < y_2 \\ \frac{y - y_2}{y_3 - y_2} + 1 \mid y \geq y_2 \end{Bmatrix} \qquad [7]$$

With this scaling, the original 9x9 matrix becomes a matrix of constants, since all the *u* and *v* are constants 0, 1 and 2, regardless of the actual values of *x* and *y*. Since the matrix is constant, its inverse is constant, and we can calculate it once, in advance, and never need to calculate it again.

When we substitute the *u*'s and *v*'s for the *x*'s and *y*'s in the matrix, we have

$$X = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 4 & 0 & 0 & 2 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 4 & 2 & 4 & 1 & 4 & 2 & 1 & 2 & 1 \\ 0 & 0 & 0 & 4 & 0 & 0 & 2 & 0 & 1 \\ 4 & 4 & 2 & 4 & 1 & 2 & 2 & 1 & 1 \\ 16 & 8 & 8 & 4 & 4 & 4 & 2 & 2 & 1 \end{bmatrix} \qquad [8]$$

and the inverse is

$$X^{-1}=\begin{bmatrix} \frac{1}{4} & -\frac{1}{2} & \frac{1}{4} & -\frac{1}{2} & 1 & -\frac{1}{2} & \frac{1}{4} & -\frac{1}{2} & \frac{1}{4} \\ -\frac{3}{4} & 1 & -\frac{1}{4} & \frac{3}{2} & -2 & \frac{1}{2} & -\frac{3}{4} & 1 & -\frac{1}{4} \\ -\frac{3}{4} & \frac{3}{2} & -\frac{3}{4} & 1 & -2 & 1 & -\frac{1}{4} & \frac{1}{2} & -\frac{1}{4} \\ \frac{1}{2} & 0 & 0 & -1 & 0 & 0 & \frac{1}{2} & 0 & 0 \\ \frac{1}{2} & -1 & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 \\ \frac{9}{4} & -3 & \frac{3}{4} & -3 & 4 & -1 & \frac{3}{4} & -1 & \frac{1}{4} \\ -\frac{3}{2} & 0 & 0 & 2 & 0 & 0 & -\frac{1}{2} & 0 & 0 \\ -\frac{3}{2} & 2 & -\frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \qquad [9]$$

and the coefficient vector solution *a* is found simply by

$$a = X^{-1} \cdot z \qquad [10]$$

Again $X^{-1}$ is a constant matrix, so the coefficients for the interpolating polynomial are found with a single matrix multiply. Another advantage to this method is that all of the elements of the inverted matrix can be represented exactly with 89/92+ BCD arithmetic, so they do not contribute to round-off error in the final result

.Once we have the polynomial coefficients, it is straight-forward to interpolate for *z* with

$$z = a \cdot u^2 \cdot v^2 + b \cdot u^2 \cdot v + c \cdot u \cdot v^2 + d \cdot u^2 + e \cdot v^2 + f \cdot u \cdot v + g \cdot u + h \cdot v + i \qquad [11]$$

This function, *intrp9z()*, implements these ideas.

```
intrpz9(xl,yl,zmat,x,y)
Func
©({xlist},{ylist},[zmatrix],x,y) 9-point z-interpolation
©Uses matrix math\im1a
©1apr01/dburkett@infinet.com

local u,v

when(x<xl[2],(x-xl[1])/(xl[2]-xl[1]),(x-xl[2])/(xl[3]-xl[2])+1)→u

when(y<yl[2],(y-yl[1])/(yl[2]-yl[1]),(y-yl[2])/(yl[3]-yl[2])+1)→v

sum(mat▸list(math\im1a*(augment(augment(zmat[1],zmat[2]),zmat[3]))ᵀ)*{u^2*v^2,u^
2*v,u*v^2,u^2,v^2,u*v,u,v,1})

EndFunc
```

Note that the matrix *im1a* (from equation [9]) must be present and stored in the *\math* folder.

The input arguments are

| | |
|---|---|
| xl | Three-element list of the table x-coordinates {x1,x2,x3} |
| yl | Three-element list of the table y-coordinates {y1,y2,y3} |
| zmat | 3x3 matrix of the table z-values |
| x | The x-value at which to interpolate |
| y | The y-value at which to interpolate |

The z-values are passed as a 3x3 matrix only because that is a convenient way for the user to enter the z-values, directly as they appear in a printed table.

The first two *when()* functions scale *x* and *y* to *u* and *v* as shown in [7]. The last line of the function calculates and returns the interpolation result. The *augment(augment(...))*$^T$ convertst the 3x3 z-matrix to a single-column vector, to perform the matrix multiplication in [10]. The coefficient solution vector is converted to a list, so that the built-in list multiplication can be used to find each term of the sum in [11].

As an example, suppose that we want to interpolate with this table at x = 0.27 and y = 0.55:

| | 0.4 | 0.5 | 0.6 |
|---|---|---|---|
| 0.1 | 0.1692 | 0.2571 | 0.3616 |
| 0.2 | 0.1987 | 0.2860 | 0.3894 |
| 0.3 | 0.2474 | 0.3335 | 0.4350 |

So we have

xl = {.1, .2, .3}
yl = {.4, .5, .6}

$$\text{zmat} = \begin{bmatrix} .1692 & .2571 & .3616 \\ .1987 & .2860 & .3894 \\ .2474 & .3335 & .4350 \end{bmatrix}$$

x = 0.27
y = 0.55

and the call to do the interpolation looks like

```
intrpz9({.1,.2,.3},{.4,.5,.6},[.1692,.2571,.3616;.1987,.286,.3894;.2474,.3335,
.435],Ø.27,Ø.55)
```

which returns 0.3664.

This program, *intrp9ui()*, provides a user interface for *intrpz9()*.

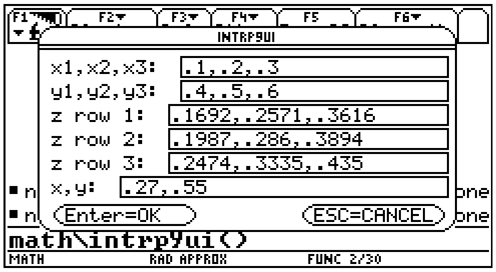
```
intrp9ui()
Prgm
©UI for intrpz9()
©calls math\intrpz9, util\rq2v, util\rq3v
©calls util\copyto_h by Samuel Stearly
©31marØ1/dburkett@infinet.com

local x123,y123,zz1,zz2,zz3,xy,res,xl,yl,zmat,x,y,z,inhome,errdisp

©Error message display program
define errdisp(msg)=Prgm
 dialog
```

```
  title "INPUT ERROR"
  text "Entry error for"
  text msg
  text "Push ENTER to try again"
 enddlog
EndPrgm

©Create lists & matrix
newlist(3)→xl
newlist(3)→yl
newmat(3,3)→zmat

©Initialize variables
"Ø,Ø,Ø"→x123
"Ø,Ø,Ø"→y123
"Ø,Ø,Ø"→zz1
"Ø,Ø,Ø"→zz2
"Ø,Ø,Ø"→zz3
"Ø,Ø"→xy
1→inhome

Lbl in1

©Prompt for input variables
dialog
 title "INTRP9UI"
 request "x1,x2,x3",x123
 request "y1,y2,y3",y123
 request "z row 1",zz1
 request "z row 2",zz2
 request "z row 3",zz3
 request "x,y",xy
enddlog
if ok=Ø:return

©Extract x-variables
util\rq3v(x123)→res
if res="ERR" then
 errdisp("x1,x2,x3")
 if ok=Ø then
  return
 else
  goto in1
 endif
else
 res[1]→xl[1]
 res[2]→xl[2]
 res[3]→xl[3]
endif

©Extract y-variables
util\rq3v(y123)→res
if res="ERR" then
 errdisp("y1,y2,y3")
 if ok=Ø then
  return
 else
  goto in1
 endif
else
 res[1]→yl[1]
 res[2]→yl[2]
 res[3]→yl[3]
endif

©Extract row 1 z-variables
util\rq3v(zz1)→res
if res="ERR" then
 errdisp("z row 1")
 if ok=Ø then
  return
 else
  goto in1
 endif
else
 res[1]→zmat[1,1]
 res[2]→zmat[1,2]
 res[3]→zmat[1,3]
endif

©Extract row 2 z-variables
util\rq3v(zz2)→res
if res="ERR" then
 errdisp("z row 2")
 if ok=Ø then
  return
 else
  goto in1
 endif
else
 res[1]→zmat[2,1]
 res[2]→zmat[2,2]
 res[3]→zmat[2,3]
endif

©Extract row 3 z-variables
util\rq3v(zz3)→res
if res="ERR" then
 errdisp("z row 3")
 if ok=Ø then
  return
 else
  goto in1
 endif
else
 res[1]→zmat[3,1]
 res[2]→zmat[3,2]
 res[3]→zmat[3,3]
endif

©Extract x,y variables
util\rq2v(xy)→res
if res="ERR" then
 errdisp("x,y")
 if ok=Ø then
  return
 else
  goto in1
 endif
else
 res[1]→x
 res[2]→y
endif
```

```
©Do the interpolation
math\intrpz9(xl,yl,zmat,x,y)→z

©Display result
dialog
 title "RESULT"
 text "x: "&string(x)
 text "y: "&string(y)
 text ""
 text "z = "&string(z)
 text ""
 dropdown "Put z in Home?",{"no","yes"},inhome
 text ""
 text "Push ENTER to interpolate again"
enddlog

©Copy result to home screen
if inhome=2
util\copyto_h(string(x)&","&string(y),z)

©Exit, or interpolate again
if ok=0: return
goto in1

EndPrgm
```

Refer to the comments at the start of the program: it requires that a few functions be installed in the *\math* and *\util* folders.

When the program is run, an input screen is shown. This screen shot shows the values entered for the example above.



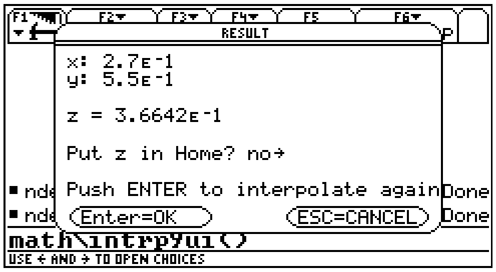
INTRP9UI

x1,x2,x3: .1,.2,.3
y1,y2,y3: .4,.5,.6
z row 1: .1692,.2571,.3616
z row 2: .1987,.286,.3894
z row 3: .2474,.3335,.435
x,y: .27,.55

math\intrp9ui()

MATH RAD APPROX FUNC 2/30

You can push [ESC] to exit the program and return to the home screen. Note that several parameters are entered in each entry field. When all the parameters are entered, the solution is calculated and displayed in this result screen:



RESULT

x: 2.7E-1
y: 5.5E-1

z = 3.6642E-1

Put z in Home? no→

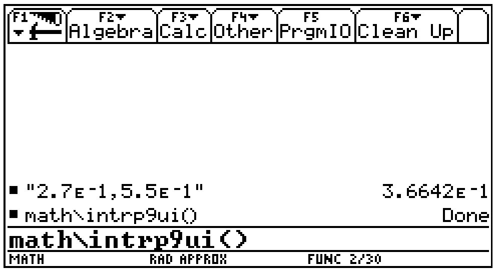
Push ENTER to interpolate again

math\intrp9ui()

USE ← AND → TO OPEN CHOICES

You can press [ENTER] to return to the input screen and do another interpolation, or press [ESC] to quit the program and return to the home screen.

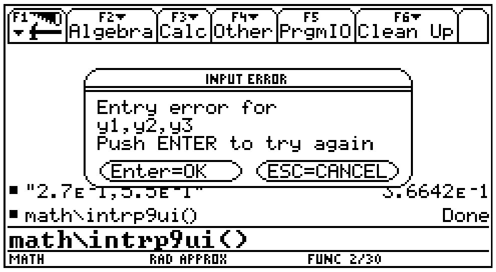
For each interpolation, you can copy the interpolation result to the home screen by selecting YES in the drop-down menu labelled *Put z in Home?*. If you choose this option, the results are copied to the home screen when you exit the program. For the example, the home screen would look like



```
■ "2.7ᴇ-1,5.5ᴇ-1"                3.6642ᴇ-1
■ math\intrp9ui()                     Done
math\intrp9ui()
MATH            RAD APPROX        FUNC 2/30
```

The result is shown in the second level of the history display. The left-hand entry shows the x- and y-coordinates as a string, and the right-hand entry is the interpolated result. This feature is made possible by Samuel Stearly's *copyto_h()* function, as described in tip [7.8].

If you forget to enter one of the commas that separate the arguments, a dialog box error message is shown:



```
INPUT ERROR
Entry error for
y1,y2,y3
Push ENTER to try again
■ "2.7ᴇ-1,5.5ᴇ-1"                3.6642ᴇ-1
■ math\intrp9ui()                     Done
math\intrp9ui()
MATH            RAD APPROX        FUNC 2/30
```

This example shows that an error has been made in entering the *y1,y2,y3* arguments. When [ENTER] is pressed, the input screen is again shown, and you can correct the error.

### [6.29] Dirac's delta (impulse) and Heaviside (step) functions

Neither of these two functions are built in to the 89/92+, but both are easily defined. Dirac's delta function, also called the impulse function, is defined as

$$\delta(x) = \left\{ \begin{array}{c} 0 \text{ when } x \neq 0 \\ \infty \text{ when } x = 0 \end{array} \right\}$$

This can be defined as a function with

```
Define delta(x)=when(x=Ø,∞,Ø)
```

The Heaviside step function is usually defined as

$$H(x) = \left\{ \begin{array}{l} 0 \text{ for } x < 0 \\ 1 \text{ for } x > 0 \end{array} \right\}$$

This is defined as a function with

```
Define H(x)=when(x<Ø,Ø,when(x=Ø,undef,1))
```

Sometimes the Heaviside function is defined as

$$H(x) = \left\{ \begin{array}{l} 0 \text{ for } x < 0 \\ \frac{1}{2} \text{ for } x = 0 \\ 1 \text{ for } x > 0 \end{array} \right\}$$

The function definition for this case is

```
Define H(x)=when(x<Ø,Ø,when(x=Ø,1/2,1))
```

### [6.30] Spherical coordinate conventions

Spherical coordinates specifiy a point or ray in 3D space with three measurements:

- The magnitude, which is the length of the ray, or the distance from the origin to the point
- The zenith angle, which is measured from the z-axis to the ray
- The azimuth angle, which is measured from the x-axis to the projection of the ray in the xy-plane.

For a picture, refer to the `▸sphere` command in your 89/92+ manual.

This definition is unambiguous. However, there is considerable ambiguity when the three measurements are grouped to specify the point. The two possibilities are

| | |
|---|---|
| (magnitude, azimuth, zenith) | *used by the 89/92+* |
| (magnitude, zenith, azimuth) | *used in other references* |

This can cause confusing results on the 89/92+, if you are used to the second specification.

Sam Jordan adds:

> *Also note that the mapping from rectangular to spherical coordinates is not exactly one-to-one since the rectangular vector [0,0,1] can be represented by any spherical vector of the form [1,<A,<0].*

In this case,

```
[Ø,Ø,1]▸sphere          returns          [1, ∠R▸Pθ(Ø,Ø), ∠Ø]
```

*(Credit to Gp and Sam Jordan)*

### [6.31] Fast 2nd-order interpolation with *QuadReg*

Linear interpolation is adequate for most table-interpolation problems. A second-order (quadratic) interpolation is more accurate, but it does require entering one more data point. The first-order (linear) interpolation uses two points to find the coefficients *a* and *b* in

$$ax + b = y$$

then calculates *y* at your desired *x*. A second-order interpolation uses three points to find *a*, *b* and *c* in

$$ax^2 + bx + c = y$$

and again finds *y* at your desired *x*. The coefficients can be found as those of a Lagrangian interpolating polynomial, but that is not built-in to the 89/92+. Quadratic regression is built-in, and it can be used to find the coefficients. The basic steps are

1. Save the x- and y-coordinates for the three points to two list variables, one for the x-coordinates, and one for the y-coordinates.
2. Execute the QuadReg command with the two lists.
3. Use the built-in system function RegEq(x) to interpolate y = f(x)

As an example, suppose that we have a table of gamma function values for x=1 to x=2, spaced every 0.04. We want to find gamma(1.895). We choose the three table x-values which bracket our desired x-value of 1.895, where the middle x-value is the one closest to the desired x-value. From the table, the three points are

| | | |
|---|---|---|
| x1 = 1.84 | y1 = 0.94261236 | |
| x2 = 1.88 | y2 = 0.95507085 | *x2=1.88 is closest to x=1.895* |
| x3 = 1.92 | y3 = 0.96877431 | |

Then these steps estimate gamma(1.895):

```
{1.84,1.88,1.92}→lx
{.94261236,.95507085,.96877431}→ly
QuadReg lx,ly
regeq(1.895)
```

which returns gamma(1.895) = 0.96006375. The actual answer is 0.960062793, for an error of about 9.6E-7. Linear interpolation results in an error of about 1.5E-4, so the 2nd-order interpolation gives several more significant digits.

Note that the *regeq*() function is a system variable that is updated when any regression command, including *QuadReg*, is executed. Once you have executed *QuadReg*, you can use *regeq()* to interpolate for additional x-values, as needed. These values need to be between *x1* and *x2*, or you are not interpolating, you are extrapolating, which is much less accurate.

This method cannot be written as a function because the *QuadReg* command only accepts global list variable names as arguments. The *QuadReg* arguments *must* be list names, and not the lists themselves.

I built this method into a program called *QuadInt(),* which is shown here:

```
quadint()
```

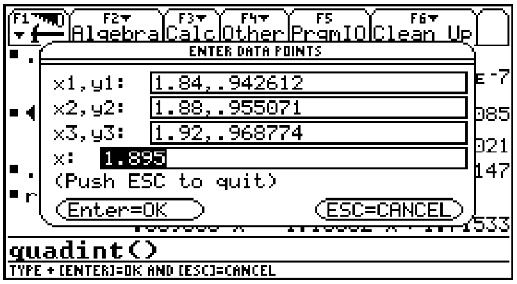
```
Prgm
©2nd-order interpolation
©15octØØ dburkett@infinet.com
©calls str2var()

local a1,b1,a2,b2,a3,b3,x1y1,x2y2,x3y3,x

© Initialize all the point variables
© (note that a1==x1, b1==y1, etc)
1→a1
2→b1
3→a2
4→b2
5→a3
6→b3
Ø→x

© Transfer control here to repeat interpolations
lbl t1

© Convert x- and y-data to pairs as strings; also convert 'x' to string
string(a1)&","&string(b1)→x1y1
string(a2)&","&string(b2)→x2y2
string(a3)&","&string(b3)→x3y3
string(x)→x

© Dialog box for user to enter xy data points
dialog
 title "ENTER DATA POINTS"
 request "x1,y1",x1y1
 request "x2,y2",x2y2
 request "x3,y3",x3y3
 request "x",x
 text "(Push ESC to quit)"
enddlog
if ok=Ø:goto exit1                    © Give user a chance to quit here

© Convert the interpolation 'x' to a number
expr(x)→x

© Convert the x1, y1 data point string to numbers
str2var(x1y1)→res
if gettype(res)="STRING":goto err1
res[1]→a1
res[2]→b1

© Convert the x2, y2 data point string to numbers
str2var(x2y2)→res
if gettype(res)="STRING":goto err1
res[1]→a2
res[2]→b2

© Convert the x3, y3 data point string to numbers
str2var(x3y3)→res
if gettype(res)="STRING":goto err1
res[1]→a3
res[2]→b3

© Build lists for QuadReg call
{a1,a2,a3}→l1
{b1,b2,b3}→l2
```

```
© Do the regression and calculate y = f(x)
quadreg l1,l2
regeq(x)→res

© Display the result
dialog
 title "QUADINT RESULT"
 text "x = "&string(x)
 text "y = "&string(res)
 text "y saved in 'res'"
 text "(Push ESC to quit)"
enddlog
if ok=0:goto exit1                    © Give user a chance to quit here
goto t1

© End up here, if user forgets to separate values with commas
lbl err1
dialog
 text "x and y values must"
 text "be separated with commas"
enddlog
goto t1                               © Go try again

© Delete global variables before exiting
lbl exit1
delvar l1,l2

EndPrgm
```

This program just provides a convenient interface for entering the xy-data points and displaying the result. Since the user might want to do several interpolations before quitting, the program loops to repeat the interpolations. The user can press [ESC] at any dialog box to quit the program. The interpolation result is saved in a global variable *res*, which can be recalled at the home screen after exiting *quadint().* The global list variables are deleted when the user quits the program.

*quadint()* calls the function *str2var()* to process the user's input. *str2var()* must be in the same folder as *quadint(),* but they may be in any folder. You must make that folder current with *setfold()* before running *quadint(). str2var()* is shown at the end of this tip.

When the program is executed, this dialog box is shown:



ENTER DATA POINTS
x1,y1: 1.84,.942612
x2,y2: 1.88,.955071
x3,y3: 1.92,.968774
x: 1.895
(Push ESC to quit)

quadint()
TYPE + [ENTER]=OK AND [ESC]=CANCEL

Each xy-data pair is entered in a single *Request* field; see tip [9.13] for more details on this. The x- and y-data values are separated with a comma. An error message is shown if the user forgets to use the comma. The picture above shows the fields filled in with the values for the example above. When all the values are entered, the user presses [ENTER] to do the interpolation. Another dialog box is shown with the interpolation result.

You can use this method with any of the built-in regression commands. You need to enter as many points as there are coefficients in the equation. For example, the *CubicReg* command fits the data to a cubic polynomial with four coefficients, so you would need to enter four points.

I mentioned that the function *str2var()* is called by *quadint().* This is *str2var():*

```
str2var(xy)
Func
©("x,y") returns {x,y}
©15octØØ dburkett@infinet.com

local s,x,y

instring(xy,",")→s              ©Find comma
if s=Ø:return "str2var err"     ©Return error string if no comma found

expr(left(xy,s-1))→x            ©Get 'x' from string
expr(right(xy,dim(xy)-s))→y     ©Get 'y' from string

return {x,y}                    ©Return list of x & y

EndFunc
```

### [6.32] Accurate approximate solutions to quadratic equations with large coefficients

This tip shows a method to calculate the approximate roots of the quadratic equation $ax^2 + bx + c =$, when *a* and *b* are very small. In this case, calculating the roots with the 'classical' solution formula results in less accurate roots. The classical solution is

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \qquad \text{or} \qquad x = \frac{2c}{-b \pm \sqrt{b^2 - 4ac}}$$

These solutions result in round-off errors for small values of *a* and *b*. A better solution is

$$q = \left(-\frac{1}{2}\right)\left(b + \mathrm{sign}(b)\sqrt{b^2 - 4ac}\right) \qquad \text{then}$$

$$x1 = \frac{q}{a} \qquad \text{and} \qquad x2 = \frac{c}{q}$$

This function, *quadrtic(),* uses these equations.

```
quadrtic(aa,bb,cc)
func
©(a,b,c) in ax^2+bx+c=Ø
©27octØØ dburkett@infinet.com
local q

when(bb≠Ø,⁻(bb+sign(bb)*√(bb*bb-4*aa*cc))/2,⁻(bb+√(bb*bb-4*aa*cc))/2)→q
{q/aa,cc/q}

Endfunc
```

To use *quadrtic(),* call it with the coefficients (a,b,c). The two roots are returned as a list. For example,

quadrtic(8,-6,1)          returns          {0.5,0.25}

To put either of the roots in the history display, use a list index of [1] or [2] as needed:

| | |
|---|---|
| {0.5,0.25}[1] | puts 0.5 in the history display |
| {0.5,0.25}[2] | puts 0.25 in the history display |

The TI92 operation mode should be set to APPROXIMATE, since the purpose of *quadrtic()* is to reduce floating-point arithmetic errors. If you need exact answers, the classical solution equations are just as good. If *quadrtic()* is run with the mode set to EXACT, it may take a very, very long time to return the answers.

For example, let a = 1E-400, b = -6 and c = 1. The classic solution gives

| | |
|---|---|
| x1 = 6E400 | y1 = 1 |
| x2 = 0 | y2 = 1 |

The improved solution equations give

| | |
|---|---|
| x1 = 6E400 | y1 = 1 |
| x2 = 0.1666... | y2 = 0 |

The improved method gives a better root for x2.

Note that the built-in *zeros()* function on the TI92+ gives the same results as the classical method.

*quadrtic()* will return complex results if the complex format is set to RECTANGULAR or POLAR. Otherwise, equations with complex results will result in an execution error.

*quadrtic()* returns these solutions for combinations of a=0, b=0 and c=0:

| | |
|---|---|
| a=0 | root 1 undefined, root 2 is solution to bx+c=0 |
| b=0 | two identical real roots, or 2 conjugate roots |
| c=0 | one root is zero |
| a=0 and b=0 | both roots are undefined; returns {undef undef} |
| b=0 and c=0 | returns 0 and 'undef' |
| a=0 and b=0 and c=0 | both roots are undefined |

### [6.33] Sine and cosine integrals

The sine and cosine integrals are functions which sometimes come up in integration and engineering applications. For example, the sine integral shows up when evaluating the radiated power of antennas, and some hypergeometric functions  The sine integral is called Si(z), and the cosine integral is called Ci(z). They are usually defined as

$$\mathrm{Si}(z) = \int_0^z \frac{\sin(t)}{t}\,dt \qquad \mathrm{Ci}(z) = \gamma + \ln(z) + \int_0^z \frac{\cos(t)-1}{t}\,dt$$

or sometimes defined as

$$\mathrm{Si}(z) = \frac{\pi}{2} - \int_z^\infty \frac{\sin(t)}{t}\,dt \qquad \mathrm{Ci}(z) = -\int_z^\infty \frac{\cos(t)}{t}\,dt$$

These symmetry relations hold:

$$\mathrm{Si}(-z) = -\mathrm{Si}(z) \qquad\qquad \mathrm{Ci}(-z) = \mathrm{Ci}(z) - i\pi \quad \text{where } 0 < \arg(z) < \pi$$

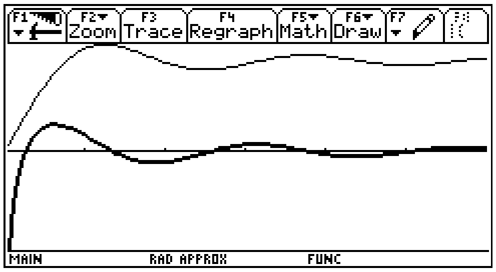
where

$\gamma = 0.577215664901533$ (Euler's constant)

Si(x) approaches $\pi/2$ as $x \to \infty$, and Ci(x) approaches 0 as $x \to \infty$. These series expansions also calculate the integrals:

$$\mathrm{Si}(z) = \sum_{n=0}^{\infty}\left[\frac{(-1)^n z^{2n+1}}{(2n+1)(2n+1)!}\right] \qquad\qquad \mathrm{Ci}(z) = \gamma + \ln(z) + \sum_{n=1}^{\infty}\left[\frac{(-1)^n z^{2n}}{2n(2n)!}\right]$$

This plot show the two functions from x = 0.1 to x = 15. The thin plot trace is Si(x), and the thick trace is Ci(x).



These functions can be calculated with the routines *Si()* and *Ci(),* shown below. These routines are not terribly accurate, but are good enough for most engineering applications. For arguments less than $\pi$ , the built-in *nint()* function is used to integrate the function definition. For arguments greater than $\pi$ , a rational approximation is used. The transition point of $\pi$ was selected more for reasons of execution time than anything else. Above that point, *nint()* takes significantly longer to execute.

The routines must be installed in a folder called */spfn*. Set the modes to Approx, and Rectangular or Polar complex before execution. Input arguments must be real numbers, not complex numbers. Note that Ci(x) returns a complex number for arguments less than zero.

Because different methods are used for different argument ranges, there is a discontinuity in the function at $x = \pi$. The discontinuity is about 2.6E-7 for Si(x), and about 1.5E-8 for Ci(x). This may not practically affect the results when simply evaluating the function, but it may be a problem if you numerically differentiate the functions, or try to find a maximum or minimum with the functions.

The rational approximations used for Si(x) and Ci(x) are based on the identities

$$\mathrm{Si}(x) = \frac{\pi}{2} - f(x)\cos(x) - g(x)\sin(x)$$

$$\mathrm{Ci}(x) = f(x)\sin(x) - g(x)\cos(x)$$

where

$$f(x) = \frac{1}{x}\left(\frac{x^8+a_1x^6+a_2x^4+a_3x^2+a_4}{x^8+b_1x^6+b_2x^4+b_3x^2+b_4}\right)+\varepsilon(x) \qquad |\varepsilon(x)| < 5\cdot 10^{-7}$$

| | |
|---|---|
| $a_1 = 38.027264$ | $b_1 = 40.021433$ |
| $a_2 = 265.187033$ | $b_2 = 322.624911$ |
| $a_3 = 335.677320$ | $b_3 = 570.236280$ |
| $a_4 = 38.102495$ | $b_4 = 157.105423$ |

$$g(x) = \frac{1}{x^2}\left(\frac{x^8+a_1x^6+a_2x^4+a_3x^2+a_4}{x^8+b_1x^6+b_2x^4+b_3x^2+b_4}\right)+\varepsilon(x) \qquad |\varepsilon(x)| < 3\cdot 10^{-7}$$

| | |
|---|---|
| $a_1 = 42.242855$ | $b_1 = 48.196927$ |
| $a_2 = 302.757865$ | $b_2 = 482.485984$ |
| $a_3 = 352.018498$ | $b_3 = 1114.978885$ |
| $a_4 = 21.821899$ | $b_4 = 449.690326$ |

All of these equations are from *Handbook of Mathematical Functions*, Abramowitz and Stegun, Dover, 1965.

The results from both of these routines are only accurate to about seven significant digits. Below $x = \pi$ , the accuracy may be as good as 11 or 12 significant digits. Notice that the maximum argument for both routines is 1E12 radians. This limit results from the use of the built-in *sin()* and *cos()* functions.

The evaluation of ci(x) and si(x) is also complicated by the fact that *nInt()* evaluates the integrals very slowly, and with poor accuracy, for arguments less than about 1E-4. To get around this, I used Taylor series expansions for the integrands near x=0, then symbolically integrated those. For the cosine integral, I use

$$\frac{\cos(t)-1}{t} = -\frac{t}{2}+\frac{t^3}{24}-\frac{t^5}{720}+\frac{t^7}{40320}+\cdots$$

and integrated this expansion to find

$$Ci(x) \simeq -\frac{x^2}{4}+\frac{x^4}{96}-\frac{x^6}{4320}+\frac{x^8}{322560}$$

Similarly, for the sine integral:

$$\frac{\sin(t)}{t} = 1-\frac{t^2}{6}+\frac{t^4}{120}-\frac{t^6}{5040}+\frac{t^8}{362880}-\frac{t^{10}}{39916800}+\cdots$$

and integrating gives

$$Si(x) \simeq x-\frac{x^3}{18}+\frac{x^5}{600}-\frac{x^7}{35280}+\frac{x^9}{3265920}-\frac{x^{11}}{439084800}$$

These approximations give good accuracy for x < 0.01.

Code listing for si(x):

```
si(x)
Func
©(x) return si(x), x real, |x|<1E12
©Must be in folder \spfn
©24jan01/dburkett@infinet.com
```

```
    if x=Ø:return Ø
    if x<Ø:return ⁻spfn\si(⁻x)

    if x>π then

    π/2-(((polyeval({1,Ø,38.Ø27264,Ø,265.187Ø33,Ø,335.67732Ø,Ø,38.1Ø2495},x))/(polye
    val({1,Ø,4Ø.Ø21433,Ø,322.624911,Ø,57Ø.23628,Ø,157.1Ø5423},x)))/(x)*cos(x)+(polye
    val({1,Ø,42.242855,Ø,3Ø2.757865,Ø,352.Ø18498,Ø,21.821899},x)/polyeval({1,Ø,48.19
    6927,Ø,482.485984,Ø,1114.978885,Ø,449.69Ø326},x))/(x^2)*sin(x))

    elseif x>.Ø1 then

    nint(sin(z)/z,z,Ø,x)

    else

    polyeval({⁻2.2774643986765ᴇ⁻9,Ø,3.Ø6192435822Ø7ᴇ⁻7,Ø,⁻2.83446712Ø1814ᴇ⁻5,Ø,1.666
    6666666667ᴇ⁻3,Ø,⁻.Ø55555555555556,Ø,1,Ø},x)

    endif

    EndFunc
```

Code listing for ci(x):

```
    ci(x)
    Func
    ©(x) return ci(x), x real, |x|<1ᴇ12
    ©Must be in folder \spfn
    ©23janØ1/dburkett@infinet.com

    if x=Ø:return undef
    if x<Ø:return spfn\ci(⁻x)+𝑖*π

    if x>π then

    (((polyeval({1,Ø,38.Ø27264,Ø,265.187Ø33,Ø,335.67732Ø,Ø,38.1Ø2495},x))/(polyeval(
    {1,Ø,4Ø.Ø21433,Ø,322.624911,Ø,57Ø.23628,Ø,157.1Ø5423},x)))/(x))*sin(x)-((polyeva
    l({1,Ø,42.242855,Ø,3Ø2.757865,Ø,352.Ø18498,Ø,21.821899},x)/polyeval({1,Ø,48.1969
    27,Ø,482.485984,Ø,1114.978885,Ø,449.69Ø326},x))/(x^2))*cos(x)

    elseif x>.Ø1 then

    nint((cos(t)-1)/t,t,Ø,x)+ln(x)+.57721566490153

    else

    ln(x)+.57721566490153+polyeval({3.1ØØ198412698ᴇ⁻6,Ø,⁻2.3148148148148ᴇ⁻4,Ø,.Ø1Ø41
    6666666667,Ø,⁻.25,Ø,Ø},x)

    endif

    EndFunc
```

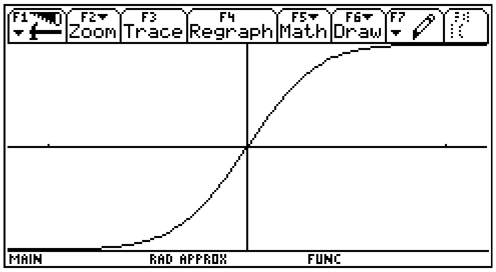
*(Thanks to Bhuvanesh for providing test data.)*

## [6.34] Error function for real arguments

The error function is one of many so-called 'special functions'. It is defined as

$$\mathrm{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$$

For x > -3 and x < 3, the error function looks like this:



erf(x) approaches 1 and -1 for large x and large -x, respectively. Also note that erf(-x) = -erf(x). The complimentary error function erfc(x) is defined as

erfc(x) = 1 - erfc(x)

This function calculates erf(x) for all real *x*.

```
erf(x)
func
©(x) error function of x
©dburkett@infinet.com 13 nov 99
©x real and -1E-999 <= x <= 1E999
©Error +/- 1E-14.

when(x<Ø,⁻erf(⁻x),when(x<.84375,x+x*polyeval({⁻2.37630166566650ᴇ⁻5,⁻5.77Ø27Ø29648
94ᴇ⁻3,⁻2.8481749575599ᴇ⁻2,⁻3.25Ø421Ø7247ᴇ⁻1,1.28379167Ø9551ᴇ⁻1},x*x)/polyeval({⁻
3.96Ø2282787754ᴇ⁻6,1.32494738ØØ432ᴇ⁻4,5.Ø813Ø62818758ᴇ⁻3,6.5Ø22249988767ᴇ⁻2,3.97
91722395916ᴇ⁻1,1},x*x),when(x<1.25,.845Ø6291151Ø47+polyeval({⁻2.1663755948688ᴇ⁻3
,3.54783Ø4325618ᴇ⁻2,⁻1.1Ø8946942824ᴇ⁻1,3.183466199Ø116ᴇ⁻1,⁻3.722Ø7876Ø357ᴇ⁻1,4.1
485611868375ᴇ⁻1,⁻2.36211856Ø7527ᴇ⁻3},x-1)/polyeval({1.1984499846799ᴇ⁻2,1.3637Ø83
912Ø29ᴇ⁻2,1.26171219Ø876ᴇ⁻1,7.1828654414196ᴇ⁻2,5.4Ø397917Ø217ᴇ⁻1,1.Ø642Ø88Ø4ØØ
84ᴇ⁻1,1},x-1),when(x<2.8571428571429,1-1/x*𝑒^(⁻x*x-.5625+polyeval({⁻9.8143293441
691,⁻8.12874355Ø63Ø7ᴇ1,⁻1.846Ø5Ø929Ø671ᴇ2,⁻1.6239666946257ᴇ2,⁻6.23753324Ø326ᴇ1,
⁻1.Ø5586262253234ᴇ1,⁻6.93858572Ø718ᴇ⁻1,⁻9.86494Ø348471ᴇ⁻3},1/x^2)/polyeval({⁻6
.Ø424415214858ᴇ⁻2,6.57Ø24977Ø3193,1.Ø8635ØØ554178ᴇ2,4.29ØØ814ØØ2757ᴇ2,6.45387271
73327ᴇ2,4.3456587747523ᴇ2,1.3765775414352ᴇ2,1.9651271667439ᴇ1,1},1/x^2)),when(x<
5.518,1-1/x*𝑒^(⁻x*x-.5625+polyeval({⁻4.835191916Ø865ᴇ2,⁻1.Ø25Ø951316111ᴇ3,⁻6.375
6644336839ᴇ2,⁻1.6Ø63638485582ᴇ2,⁻1.7757954917755ᴇ1,⁻7.99283237680 52ᴇ⁻1,⁻9.864942
9247ØØ1ᴇ⁻3},1/x^2)/polyeval({⁻2.244Ø952446586ᴇ1,4.74528541 2Ø696ᴇ2,2.553Ø5Ø4Ø6433
2ᴇ3,3.1998582195Ø86ᴇ3,1.536729586Ø844ᴇ3,3.2579251299657ᴇ2,3.Ø338Ø6Ø743482ᴇ1,1},1
/x^2)),1)))))

Endfunc
```

This function is accurate to full machine precision for all arguments. The algorithm is taken from module ERF in the FDLIBM package. This link provides the algorithm details:

*http://gams.nist.gov/serve.cgi/Module/FDLIBM/ERF/13299/*

While the function may look complicated, it is really nothing more than a series of nested *when()* functions to select the appropriate estimating method for value the input *x* argument. *erf()* calls itself for x<0; there is only one level of recursion.

These references have more discussion of the error function:

*Handbook of Mathematical Functions*, Abramowitz and Stegun, Dover, 1965.

*Special Functions and Their Applications*, Lebedev, Dover, 1972.

### [6.35] Cumulative normal distribution and inverse

The cumulative normal (or Gaussian) distribution is defined as

$$\Pr\{X \le x\} = p(x) = \frac{1}{\sigma\sqrt{2\pi}} \int_{-\infty}^{x} e^{\frac{(t-m)^2}{2\sigma^2}} dt \qquad [1]$$

where *m* is the mean and $\sigma^2$ is the variance, so $\sigma$ is the standard deviation. In other words, this integral is the area under the probability function to the left of *x*. This is also sometimes called the 'lower tail' of the distribution. When the mean is 0 and the variance is 1, this is called the *standard* cumulative normal distribution.

You can find the cumulative normal distribution by integrating equation [1] with the *nint()* function. The function below, *cndint(),* does this.

```
cndint(x,m,s)
Func
©(x,mean,std dev) find CND with integration
©29octØØ/dburkett@infinet.com
1/(s*√(2*π))*nint(e^(⁻.5*((t-m)/s)^2),t,⁻∞,x)
Endfunc
```

For x = 0, mean = 0 and standard deviation = 1, *cndint()* returns the probability p = 0.49999999952549 in about 7.5 seconds on a HW2 92+ with AMS 2.05. The answer should be p = 0.5. For x = 6, mean = 0 and standard deviation = 1, *cndint()* returns p = 0.99999999853883 in about 10.5 seconds. This answer is in error by about -5E-10. We can write functions that are much faster and somewhat more accurate.

It turns out that the cumulative normal distribution (called CND from now on) can be found efficiently with the error function erf(x). First, this identity relates the standard CND to the general CND:

$$\Pr\{X \le x\} = P\left(\frac{x-m}{\sigma}\right) \qquad [2]$$

where P(x) is the standard CND. This converts a distribution with mean *m* and standard deviation $\sigma$ to the standard CND. Next, this identity relates P(x) to erf(x):

$$\mathrm{erf}(x) = 2 \cdot P(x\sqrt{2}) - 1 \qquad [3]$$

which can be solved for

$$p(x)=\frac{\mathrm{erf}\left(\frac{x}{\sqrt{2}}\right)+1}{2} \qquad [4]$$

This can be combined with [2] to yield a simple function to find the general CND, like this:

```
cnd(x,m,s)
func
©(x,m,s) find cum norm dist at x, mean m, std dev s
©Find cumulative distribution at x, with mean m and standard deviation s
©dburkett@infinet.com 8jun00

(erf(((x-m)/s)/1.4142135623731)+1)/2

Endfunc
```

This function calls *erf(),* described in tip [6.34], which must be in the same folder as *cnd().* That folder must be the current folder.

As an example, find the standard cumulative normal distribution for x = 1:

`cnd(1,0,1)` returns 0.841345

To find the cumulative normal distribution for at x = 12, for a distribution with a mean of 10 and a standard deviation of 2.5, use

`cnd(12,10,2.5)` which returns 0.788145

For some problems you need the 'upper tail' distribution, which is the area to the *right* of the sample, or the probability that the random variable is greater than some *x*. This can be found with *cnd(),* as expected: just use 1 - *cnd().* For example, to find the upper tail normal distribution for x = 2, mean = 0 and standard deviation = 1, use

`1-cnd(2,0,1)` which returns 0.0227501319482

For some other problems, you might need to find the probability that the random variable is between two limits x1 and x2, where x2 > x1. In this case, use

```
cnd(x2,m,s) - cnd(x1,m,s)
```

where *m* is the mean and *s* is the standard deviation. For example, to find the probability that the random variable is between 1.6 and 2.2, for a distribution with mean = 2 and standard deviation = 0.5, use

`cnd(2.2,2,.5) - cnd(1.6,2,.5)` which returns 0.443566

For real problems, the mean and standard deviation will probably have more digits, and you can save a little typing by entering this example like this, instead:

```
cnd(2.2,m,s)-cnd(1.6,m,s)|m=2 and s=.5
```

The CND function is available in the free TI *Statistics and List Editor* (SLE) flash application. However, if you don't need all the additional functions that the SLE includes, you may prefer to use these two simple functions, instead of filling up your flash memory with the 300K SLE. In addition, the function shown here is more accurate than that included in the SLE application. For the upper-tail distribution with x = 2, mean = 0 and standard deviation = 2,the SLE returns a result of .022750062014, which is in error by about 8E-8.

It is occasionally necessary to find the inverse of distribution function, that is, to find the value of the random variable *x*, given a probability, a mean and a standard deviation. There is no closed-form solution to this problem. One possiblity is to use the built-in *nSolve()* function with *cnd().* For example, suppose you know that the distribution has a mean of 7, a standard deviation of 1. What is the value of the random variable *x* for which the probability is 0.2? Use *nSolve()* like this:

```
nSolve(cnd(x,7,1)=Ø.2,x)
```

which returns x = 6.158379 in about 19 seconds on a HW2 92+.. You can (and should) check this solution with

`cnd(6.158379,7,1)` which returns 0.2, as hoped.

The execution time can be considerably reduced by providing *nSolve()* with a good initial guess for the value of *x*. If the initial guess is good enough, we don't need to use *nSolve()* at all, and the result is returned very quickly.

For the standard CND, with mean = 0 and standard deviation = 1, Odeh and Evans give this estimating function for $x_p$ for a given probability *p*:

$$x_p = f(t) = t + \frac{p_4t^4+p_3t^3+p_2t^2+p_1t+p_0}{q_4t^4+q_3t^3+q_2t^2+q_1t+q_0} \quad \text{and} \quad t = \sqrt{-2\ln(p)} \quad \text{and } 10^{-20} < p < 0.5 \qquad [5]$$

and

| | |
|---|---|
| $p_0$ = -0.32223 24310 88 | $q_0$ = 0.09934 84626 060 |
| $p_1$ = -1.0 | $q_1$ = 0.58858 15704 95 |
| $p_2$ = -0.34224 20885 47 | $q_2$ = 0.53110 34623 66 |
| $p_3$ = -0.02042 31202 45 | $q_3$ = 0.10353 77528 50 |
| $p_4$ = -0.45364 22101 48 E-4 | $q_4$ = 0.38560 70063 4 E-2 |

For the general case in with mean *m* and standard deviation *s*, we have

$$x_p = s \cdot f(t) + m \qquad [6]$$

using the identity in equation [2]. Finally, we use the symmetry of the normal distribution about the mean to account for p≥ 0.5, like this:

$$x_p = [\mathrm{sgn}(p-0.5)][s \cdot f(t)] + m \qquad [7]$$

Here, sgn(x) is a 'sign' function such that

$$\mathrm{sgn}(x) = \begin{Bmatrix} -1 \text{ for } x < 0 \\ 1 \text{ for } x \geq 0 \end{Bmatrix} \qquad [8]$$

We cannot use the built-in *sign()* function, because it returns the expression 'sign(0)' for x=0, and we want 1. This is fixed with a *when()* function:

```
sgn(x) = when(x≠Ø,sign(x),1)
```

The function shown below, *cndif(),* implements the algorithm just described to find the inverse of the cumulative distribution function.

```
cndif(p,m,s)
func
©(probability,mean,std dev) inverse CND
©Inverse cumulative normal distribution
©Returns x with probability that X≤x
©Fast version.
©29octØØ/dburkett@infinet.com

local r,t

p→r
if p>.5:1-r→r
√(⁻2*ln(r))→t

when(p≠.5,sign(p-.5),1)*s*(t+(polyeval({⁻.453642210148ᴇ⁻4,⁻.2Ø4231210245ᴇ⁻1,⁻.34
2242Ø88547,⁻1,⁻.322232431Ø88},t)/polyeval({.3856Ø7ØØ634ᴇ⁻2,.1Ø353775285,.5311Ø34
62366,.588581570495,.993484626Ø6ᴇ⁻1},t)))+m

Endfunc
```

For example, find *x* with a probability of 0.25, for a distribution with mean = 20 and standard deviation = 0.15:

```
cndif(.25,2Ø,.15)
```
returns x = 19.898826537593

We can check this answer with *cnd():*

```
cnd(19.898826537593,2Ø,.15)
```
returns p = 0.25000000025933

*cndif()* is fast, since it simply uses two *polyeval()* functions to calculate the estimating rational polynomial. The reference for Odeh and Evans' estimating function claims about 7 significant digits of accuracy. If you need more accuracy than this, you can use this function, *cndi():*

```
cndi(p,m,s)
func
©(probability,mean,std dev) inverse CND
©Inverse cumulative normal distribution
©Returns x with p=probability that X≤x
©Accurate, slow version.
©29octØØ/dburkett@infinet.com

local r,t

p→r
if p>.5:1-r→r
√(⁻2*ln(r))→t

when(p<.999999713348,nsolve(cnd(x,m,s)=p,x=when(p≠.5,sign(p-.5),1)*s*(t+(polyeva
l({⁻.453642210148ᴇ⁻4,⁻.2Ø4231210245ᴇ⁻1,⁻.342242Ø88547,⁻1,⁻.322232431Ø88},t)/poly
eval({.3856Ø7ØØ634ᴇ⁻2,.1Ø353775285,.5311Ø3462366,.588581570495,.993484626Ø6ᴇ⁻1},
t)))+m),when(p≠.5,sign(p-.5),1)*s*(t+(polyeval({⁻.453642210148ᴇ⁻4,⁻.2Ø4231210245
ᴇ⁻1,⁻.342242088547,⁻1,⁻.322232431088},t)/polyeval({.38560700634ᴇ⁻2,.10353775285,
.531103462366,.588581570495,.99348462606ᴇ⁻1},t)))+m)

Endfunc
```

*cndi()* uses *nSolve()* to find *x*, with an initial guess found by Odeh & Evans' expression. However, if p > 0.999999713348, *cndi()* just uses Odeh & Evan's expression. This value of *p* corresponds to a standard deviation of 5. Above this limit, *nSolve()* does not improve the accuracy, and takes much longer.

*cndi()* typically executes in a few seconds. While not extremely impressive, this is faster than simply using *nSolve()* with no initial guess.

These are the references I used to develop the functions:

*Handbook of Mathematical Functions*, Abramowitz and Stegun, Dover, 1965. The standard normal probability distribution, and the transformation to the general normal distribution, are described on p931. The relation to the error function is shown on p934.

*Statistical Computing*, Kennedy and Gentle, Marcel Dekker, 1980. Odeh and Evans' approximation for the inverse cumulative normal distribution is shown on p95.

### [6.36] Integration may return 'undef' for identical integration limits

By definition,

$$\int_a^a f(x)dx = 0$$

for any *a* and *f(x)*. The TI-89/92+ evaluate such 'zero-width' integrals correctly with either *nInt()* or `∫()`, unless *f(x)* cannot be evaluated at *a*. For example, these functions all return zero:

| | | |
|---|---|---|
| `∫(1,x,0,0)` | or | nInt(1,x,0,0) |
| `∫(x,x,0,0)` | or | nInt(x,x,0,0) |
| `∫(sin(x),x,0,0)` | or | nInt(sin(x),x,0,0) |

However, if f(a) is undefined, then these expressions return *undef*:

| | | |
|---|---|---|
| `∫(1/x,x,0,0)` | or | nInt(1,x,0,0) |
| `∫(sin(x),x,0,0)` | or | nInt(1,x,0,0) |

If the integration limits are specified symbolically instead of numerically, then `∫()` correctly returns zero, but *nInt()* returns itself:

| | |
|---|---|
| `∫(1/x,x,0,0)` | returns 0 |
| `nInt(1/x,x,0,0)` | returns `nInt(1/x,x,0,0)` |

This behavior is not a problem for manual calculations because you would never bother to evaluate an integral with identical limits. However, it could be a problem if you evaluate integrals within a program or function. If so, test the integration limits for equality before evaluating the integral. If they are equal, return zero. One method is

```
when(a=b,Ø,∫(1/x,x,a,b))
```

### [6.37] Random number generator algorithm

For some simulations and Monte Carlo numerical methods, it is important that the random number generator have a sufficiently long period. I emailed TI-cares concerning this. Here is the response:

> *The TI-89 and TI-92 Plus all use the same random number generator. Rand() creates random numbers between 0 and 1.*
>
> *The specific algorithm is described in a research contribution by L'ecuyer [1].*
>
> *Essentially, this algorithm combines two multiplicative linear congruential generators to form a single generator with extremely long period. Our particular implementation of random number generator uses the first two sets of coefficients from Table III on page 747 of reference [1].*
>
> *This generator was submitted to several tests: equidistribution test, serial test, gap test, poker test, coupon's collector test, permutation test, runs-up test, maximum-of-t test, and collision test. The random number generator we implemented in these units passes all these tests for randomness.*
>
> *References.*
> *1. L'ecuyer, P. Efficient and Portable Combined Random Number Generators. Communications of the ACM, Vol. 31, Number 6 (June 1988), pp. 742-749, 774.*

### [6.38] Finance Flash Application function finds days between two dates

If you have installed the TI Finance flash application, you can use the *dbd()* function to find the number of days between two dates, but both dates must be between the years 1950 and 2049. To call the function, use

```
tifnance.dbd(date1,date2)
```

You can type this in the command line, or use [CATALOG] [F3] to paste it to the command line. The format for the dates is either MM.DDYY or DDMM.YY. MM is the month, DD is the day of the month, and YY is the last two digits of the year. Note that the decimal point distinguishes between the two formats. For example, to find the number of days between January 15, 2001, and December 17, 2002, use

```
tifnance.dbd(1.15Ø1,12.17Ø2)
```

which returns 701 days.

### [6.39] Convert equations to a parameterized form

A function f(x,y) = 0 can be 'parameterized' by converting it into two other functions x(t) and y(t), where *t* is the parameter. This functions x(t) and y(t) are not necessarily unique; there may be many functions that give the same result f(x,y). This function applies one possible parameterization of *x* and *y*, namely t = y/x.

```
parcurve(f,x,y)
Func
©ParCurve(f,x,y) parametrizes curve f(x,y)=0 as x(t),y(t)
©Bhuvanesh Bhatt
expr("solve("&string(f=y-t*x)&" and y-t*x=0,{"&string(x)&","&string(y)&"})")
EndFunc
```

For example, to find a parametric expression for

$$x^3 + x^2 - y^2 = 0$$

use the call

```
parcurve(x^3+x^2-y^2,x,y)
```

which returns

$$x = t^2 - 1 \text{ and } y = \left(t^2 - 1\right) \cdot t \text{ or } x = 0 \text{ and } y = 0$$

The first two equations are the useful parameterizations; the second two are extraneous solutions.

*(Credit to Bhuvanesh Bhatt)*

### [6.40] Write functions with multiple input argument types

Many built-in functions operate on numbers, expressions and lists. For example,

| | | |
|---|---|---|
| `sin(a)` | returns | `sin(a)` |
| `sin(.2)` | returns | `0.1987` |
| `sin({0,.2})` | returns | `{0, 0.1987}` |

This is very convenient in that you can use a single function to handle various argument types. You can accomplish this convenience with your own functions, as shown in this demonstration function:

```
f1demo(xx)
Func
©Demo program to evaluate function of expression, list or matrix.
©29mar01/dburkett@infinet.com

local f1,xxt,j,k,r,xxr,xxc

©Define the function to evaluate
define f1(x)=func
 ln(x)
endfunc

©Get the input argument type
gettype(xx)→xxt

©Evaluate the function
if xxt="NUM" or xxt="EXPR" or xxt="VAR" then
 return f1(xx)
elseif xxt="LIST" then
 return seq(f1(xx[j]),j,1,dim(xx))
elseif xxt="MAT" then
```

```
 rowdim(xx)→xxr
 coldim(xx)→xxc
 newmat(xxr,xxc)→r
 for j,1,xxr
  for k,1,xxc
   f1(xx[j,k])→r[j,k]
  endfor
 endfor
 return r
else
 return "f1demo: type error"
endif

EndFunc
```

This function accepts arguments that may be expressions, numbers, lists or matrices. If the argument is a list or matrix, the function is applied to each element. The function to be evaluated is defined as a local function *f1()*; as an example, I use the *ln*() function. Your functions will probably be more complicated.

The argument type is determined and saved in variable *xxt*. If the argument is a number, expression or variable name, the argument is evaluated by the function and returned. If the argument is a list, the *seq()* function is used to evaluate each list element, and return the results as a list. If the argument is a matrix, two nested *For ... EndFor* loops are used to evaluate the function of each matrix element, and the matrix is returned.

If the argument type is not supported, *f1demo()* returns the text string *"f1demo: type error".* The calling program can test the type of the returned result and determine if an error occurred.

### [6.41] Integration error in AMS 2.05

AMS 2.05 on TI89/92+ calculators can return incorrect results for indefinite and definite integrals which include, in the integrand, the expression

$$\sqrt{(ax+b)^n}$$

where *x* is the integration variable and *a*, *b* and *n* are constants. This bug has been reported by Damien Cassou and others.

One solution is to use the exponential form for the integrand, instead of the square root operator.

For example, in Auto mode,

$$\int \sqrt{(7x+4)^3}\,dx \qquad \text{returns} \qquad \frac{2(7x+4)\sqrt{(7x+4)^3}}{5} \qquad [1]$$

while the correct result should be

$$\frac{2(7x+4)\sqrt{(7x+4)^3}}{35}$$

Another example of an incorrect result is

$$\int \sqrt{(2x-3)^7}\,dx \qquad \text{which returns} \qquad \frac{2(2x-3)\sqrt{(2x-3)^7}}{7} \qquad [2]$$

but the correct result is $\frac{(2x-3)\sqrt{(2x-3)^7}}{9}$

However, if symbolic values are used for the constants, the correct result is returned, that is

$\int \sqrt{(ax+b)^n}\,dx$ returns $\frac{2(ax+b)^{\frac{n}{2}+1}}{a(n+2)}$ [3]

If Approx mode is used instead of Auto mode, examples [1] to [3] return correct results.

The problem also exists with definite integrals. This is shown with a definite integral of example [1] above:

$\int_2^9 \sqrt{(7x+4)^3}\,dx$ returns $\frac{8978\sqrt{67}}{5} - \frac{1944\sqrt{2}}{5}$ [4]

but the correct result is $\frac{8978\sqrt{67}}{35} - \frac{1944\sqrt{2}}{35}$

In Approx mode the correct result of about 2021.11 is returned.

Correct results can be obtained in Auto mode by writing the integrand in exponential form, instead of using the square root operator. Applying this to example [1]:

$\int (7x+4)^{\frac{3}{2}}$ correctly returns $\frac{2(7x+4)^{\frac{5}{2}}}{35}$ [5]

and for the definite integral of example [4]:

$\int_2^9 \sqrt{(7x+4)^3}\,dx$ correctly returns $\frac{8978\sqrt{67}}{35} - \frac{1944\sqrt{2}}{35}$ [6]

Incorrect results will be obtained if you use substitution (the "with" operator | ) for *n*, for example

$\int \sqrt{(7x+4)^n}\,dx \mid n=3$ returns the incorrect result of [1]

but if you evaluate the expression without the constraint *n=3*, and then apply the constraint to that result, the correct integral is returned.

A similar problem also occurs with integrands of the form

$\frac{1}{\sqrt{(ax+b)^n}}$ [7]

For integrals of this form, the correct result is returned if$n \geq 10$ or$n \leq -10$ .

The built-in differential equation solver can fail with arguments of this type, too.

*(Credit to Gary Wardall, Damien Cassou, Kevin Kofler, and ES*)

### [6.42] stdDev() and variance() find sample (not population) statistics

The variance and standard deviation statistics measure the deviation of data about the mean. If the data consists of the entire population of *n* elements, the statistics are

Population standard deviation: $\sigma = \sqrt{\dfrac{\sum_{i=1}^{n}\left(x_i-\mu\right)^2}{n}}$ , where $\mu$ is the population mean

Population variance: $\sigma^2$

Sample standard deviation: $s = \sqrt{\dfrac{\sum_{i=1}^{n}\left(x_i-xm\right)^2}{n-1}}$ , where $x_m$ is the sample mean

Sample variance: $s^2$

For example, suppose I have the data {5,5,6,7,8,9}. Then$\sigma$ = 1.4907 and s = 1.6330.

The built-in command *OneVar* calculates both the population and sample standard deviations, however, only the sample standard deviation (called *Sx*) is displayed with the *ShowStat* command. To display the population standard deviation (called σx) after using *OneVar*, you must manually recall σx as follows:

TI-89: [DIAMOND] [ ( ] [alpha] [S] [x] [ENTER]
TI-92+: [2ND] [G] [S] [x] [ENTER]

The *stdDev()* and *variance()* functions of the 89/92+ return the sample statistics, not the population statistics. However, these functions can be used to calculate the population statistics if the results are multiplied by a suitable correction factor:

$\sigma = s\sqrt{\frac{n-1}{n}}$ and $\sigma^2 = s^2\left(\frac{n-1}{n}\right)$

which can be written as

$\sigma = s\sqrt{1-\frac{1}{n}}$ and $\sigma^2 = s^2\left(1-\frac{1}{n}\right)$

so that *n* is referenced only once, instead of twice; this slightly simplifies the calculation. The correction factor is found by equating the equivalent expressions for σ and s. Let

$$SSx = \sum_{i=1}^{n}\left(x_i - m\right)^2$$

where *m* is either the population mean or the sample mean, then

$\sigma^2 = \frac{SSx}{n}$ and $s^2 = \frac{SSx}{n-1}$ or $SSx = s^2 \cdot (n-1)$ then

$$\sigma^2 = s^2\left(\frac{n-1}{n}\right) \qquad \text{and} \qquad \sigma = s\sqrt{\frac{n-1}{n}}$$

The last line shows the identities we want, in which the desired population variance and standard deviation are functions of the sample variance and standard deviation.

If desired, these definitions can be coded as simple functions:

*Population standard deviation:*

```
stddevp(x)
Func
©(list) population std dev of list
©28aprØ1/dburkett@infinet.com

stddev(x)*√(1-1/dim(x))

EndFunc
```

*Population variance:*

```
variancp(x)
Func
©(list) population variance of list
©28aprØ1/dburkett@infinet.com

variance(x)*(1-1/dim(x))

EndFunc
```

The built-in *stddev()* and *variance()* functions can both accept an optional second argument which is a list of the frequencies of the first list elements. This feature is easily added with these two functions:

*Population standard deviation, with frequencies:*

```
stddevpf(x,f)
Func
©(list,freq) population std dev of list with element frequency 'freq'
©28aprØ1/dburkett@infinet.com

stddev(x,f)*√(1-1/sum(f))

EndFunc
```

*Population variance, with frequencies:*

```
varianpf(x,f)
Func
©(list,freq) population variance of list with element frequency 'freq'
©28aprØ1/dburkett@infinet.com

variance(x,f)*(1-1/sum(f))

EndFunc
```

In these last two functions, the number of data points is found by summing the elements of the frequency list.

### [6.43] Dot product (scalar product) for complex vectors

The built-in *dotp*() function can be used to find the scalar dot product for two complex vectors, but its behavior may be puzzling. The vector dot product of **a** and **b** is defined as

$$\mathbf{a} \cdot \mathbf{b} = \|\mathbf{a}\|_2 \|\mathbf{b}\|_2 \cos(\theta) \qquad [1]$$

where θ is the angle between the two vectors, and

$$\|\mathbf{a}\|_2 = \sqrt{\sum_{i=1}^{n} a_i^2}$$

which is the $L_2$ (Euclidean) norm. Suppose that we want to find the dot product of a vector and itself, or

$$\mathbf{a} \cdot \mathbf{a} = \|\mathbf{a}\|_2 \|\mathbf{a}\|_2 \cos(\theta) = \|\mathbf{a}\|_2^2 \cos(0) = \|\mathbf{a}\|_2^2$$

In other words, the dot product of a vector with itself must be real and non-negative. *dotp*() correctly returns this result. Some users claim that the 89/92+ calculate this function in error, and that the complex conjugate of the second argument must be taken, to get the 'correct' result. However, suppose we apply this technique to the vector [1+i, 1+i], where 'i' is the square root of -1. Then

```
dotp([1+i],[1+i]) = 4
```

as expected, but

```
dotp([1+i],conj([1+i])) = 4i
```

which must be wrong, because the result must be real and non-negative. Some of this confusion may result because there are actually two working definitions of dot product. In linear algebra, optimization and computer science, the inner product is defined as

$$\mathbf{x}^T \mathbf{y} = \sum_{i=1}^{n} x_i \cdot y_i \qquad [2]$$

While this definition provides consistent results in real linear algebra and optimization, it cannot be used for the complex vector dot product, because it fails in the case of finding the dot product of a vector with itself.

For real-valued vectors, there is no ambiguity, and *dotp*() returns the correct results. Note also that *dotp*() can accept lists as arguments, and for complex elements it also implicitly uses the definition in [1] above, not [2].

*(Credit to Bhuvanesh Bhatt and Fabrizio)*

### [6.44] Nest min() in max() to limit an argument

You can limit an argument to two bounds *upper* and *lower*, with this expression:

```
max(lower,min(x,upper))
```

For example, suppose we have a calculation in which we calculate the real inverse sine (arcsine) of the result of a previous calculation. Round-off errors in that calculation may result in an argument *x* whose absolute value is slightly greater than 1, so finding the inverse sine of the argument will result in an error, or a complex result. We can avoid this by finding the inverse sine of

```
max(-1,min(x,1))
```

If *x* is between -1 and 1, inclusive, then the expression returns *x*. If *x* is less than -1, the expression returns -1, and if *x* is greater than 1, the expression returns 1. This same result can be obtained with an *If ... EndIf* structure, or as a *when()* function, but this form is more concise.

### [6.45] Determine if a point is inside a triangle

This tip is more application-specific than most, but the idea is so clever that I had to include it. Suppose that we have a triangle defined by the three points A, B and C, with rectangular coordinates ($x_a$,$y_a$), ($x_b$,$y_b$) and ($x_c$,$y_c$). We have another point P at (x,y), and we want to know if P is inside the triangle ABC. In general, if P is in ABC, then the area of ABC will equal the sum of the areas of the three triangles PBC, APC and ABP. If P is outside ABC, then the area of ABC is less than the sum of the areas of the three triangles formed by A, B, C and P. To find the area of a triangle with points $P_1$, $P_2$ and $P_3$, we can use

$$\text{area}(P_1P_2P_3) = \frac{\left\| \begin{matrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{matrix} \right\|}{2} = \text{abs}\left(\frac{x_1y_2+x_3y_1+x_2y_3-x_1y_3-x_2y_1-x_3y_2}{2}\right)$$

where the double bars indicate the absolute value of the determinant of the matrix. By using the absolute value, we ensure that the area is positive, regardless of the ordering of the three points. This function implements the idea:

```
PntTrian(px,py,ax,ay,bx,by,cx,cy,tol)
Func
©(px,py,ax,ay,bx,by,cx,cy,tol) return True if point P is in triangle ABC
©25may01/dburkett@infinet.com

local tArea,at

©Define local function to find triangle area
Define tArea(xx1,yy1,xx2,yy2,xx3,yy3)=Func
abs(xx1*yy2-xx1*yy3-xx2*yy1+xx2*yy3+xx3*yy1-xx3*yy2)/2
EndFunc

©Find area difference & return result

tArea(ax,ay,bx,by,cx,cy)→at
```

```
return
abs((at-(tArea(px,py,bx,by,cx,cy)+tArea(ax,ay,px,py,cx,cy)+tArea(ax,ay,bx,by,px,
py)))/at)≤tol

EndFunc
```

The function returns *true* if point (*px*,*py*) is within the triangle defined by the points (*ax*,*ay*), (*bx*,*by*) and (*cx*,*cy*), otherwise it returns *false*. *tol* is a small positive number which specifies the tolerance at which the two areas will be considered equal. If you are using Exact mode, you may set *tol* = 0. If you are using the function in Approx mode, with floating point coordinate arguments, set *tol* to specify the precision at which you want to consider the two areas equal. This will depend on your application. If, for example, you want to assume that the point P is in triangle ABC if the two areas are the same to six significant digits, then set *tol* = 1E-6. A safe minimum value for *tol* is 1E-12, since the 89/92+ results are accurate to about 12 significant digits.

Note that I use a relative tolerance, not an absolute tolerance. This is necessary because the point coordinates may have a wide range. For example, suppose that the three triangle points are (0,0), (0,2E20) and (1E20,1E20), and we want to determine if point P = (-1E10,-1E10) is in the triangle. The area of ABC is 1E40, and the sum of the areas of the three triangles including P is 1.0000000002E40. The absolute difference between the two areas is 2E30, but the relative difference is 2E-10. If we call *PntTrian*() with *tol* = 1E-12, it will correctly return *false* in this case.

Disregarding the tolerance, points at the triangle vertices and on the lines joining the vertices are considered inside the triangle.

As an example, suppose we have a triangle with vertices at (0,0), (7,0) and (4,4). Which of the points (4,1), (5,3) and (2,2) are within the triangle? We will use exact mode and *tol* = 0, so the function calls are

| | |
|---|---|
| `PntTrian(4,1,Ø,Ø,7,Ø,4,4,Ø)` | returns *true* |
| `PntTrian(5,3,Ø,Ø,7,Ø,4,4,Ø)` | returns *false* |
| `PntTrian(2,2,Ø,Ø,7,Ø,4,4,Ø)` | returns *true* |

So the first and third points are in the triangle, but the second point is not. Note that the third point is on a triangle edge.

The idea for this tip was taken from a Design Idea in the August 3, 2000 issue of *Electronic Design News* magazine, titled *Algorithm tests for point location*, by Lawrence Arendt, of the Mannitoba HVDC Research Centre, Winnipeg, Canada.

### [6.46] solve() and nsolve() ignore constraints with local variables

*(Applies to AMS 2.05)*

Constraint expressions are used with *solve()* and *nsolve()* to limit the solution range for the independent variable. This shortens the solution time, and also narrows the search to the desired root if the function has more than one. For example, the constraint expression "|x>1 and x<2" in this example

```
nsolve(f(x)=Ø,x)|x>1 and x<2
```

should return a single solution between x=1 and x=2.

In July 2001, David Dannenmiller posted an *nsolve()* example to the 89/92+ discussion group, where the constraints were ignored and *nsolve()* returned a root outside of the solution bounds. It turns out that David found a particular constraint expression which causes *nsolve()* to return a solution which is a root, but is outside of the solution bounds. In particular, the constraint is expressed with a local variable, and the constraint bound is expressed as a>x instead of x<a, where 'a' is a local variable.

So, if you use *solve()* or *nsolve()* in a program or function, and you use local variables in the constraint expression, be sure to list the independent variable *first* in the contraint expression, to the left of the conditional operator:

Do this: `nsolve(f(x)=Ø,x)|x<a and x<b`

*Don't* do this: `nsolve(f(x)=Ø,x)|a>x and x<b`

As an example, consider this function:

$$f(x) = x^3 - 6x^2 + 11x - 6 + \sin(3x)$$

which has these three real roots when f(x)=0:

x1 = 0.71379958487236
x2 = 1.1085563021699
x3 = 2.14042730947

Suppose we use *nsolve()* in a function to find the third root, x3. The function might look like this:

```
f()
Func
local a,b

2→a
2.2→b

return nsolve(x^3-6*x^2+11*x-6+sin(3*x)=Ø,x)|x>a and x<b

EndFunc
```

This is a contrived example because you would not actually use such a function, to solve a single function for a single root. A more realistic application would solve a more complicated function with variable coefficients, and the constraint limits *a* and *b* would be found by some estimating method.

As shown, the constraint is |x>a and x<b. This table shows the returned results for various other constraint expressions

| Constraint | Result | Comment |
|---|---|---|
| x>a and x<b | x3 | |
| a<x and x<b | x1 | Wrong! |
| a<x and b>x | x1 | Wrong! |
| x>a and b>x | x3 | |

Note that if numeric limits are used, instead of local variables, then *nsolve()* returns the correct results regardless of the constraint ordering. For example, *nsolve()* will return the correct root x3 if the contraint is |2<x and x<2.2.

*solve()* shows the same behavior as *nsolve()*, except that it takes much longer to return the incorrect roots, and the correct root is often included as one of multiple solutions.

### [6.47] Unsigned infinity displays as *undef*

'Unsigned infinity' is sometimes used in math textbooks to indicate a limit which is +∞ on one side of the evaluation point, and -∞ on the other side of the evaluation point. This is usually indicated as ±∞. The 89/92+ return *undef* (undefined) instead of unsigned infinity, because the majority of courses for which the calculator was designed do not address unsigned infinity. However, the calculator internally carries unsigned infinity as a result.

This behavior can be demonstrated as :

1/0 returns *undef*
abs(1/0) returns ∞, since the absolute value of unsigned infinity is +∞

For another example, consider

$\tan\left(\frac{\pi}{2}\right)$ returns *undef*, but

$\left[\tan\left(\frac{\pi}{2}\right)\right]^2$ returns ∞, since ∞ is the square of unsigned infinity

*(Credit to Paul King)*

### [6.48] Use R▸P$\theta$() for four-quadrant arc tangent function

The built-in arc tangent function, *tan$^{-1}$()*, cannot return the correct quadrant of an angle specified by x- and y-coordinates, because the argument does not contain enough information. Suppose we want the angle between the x-axis and a ray with origin (0,0) and passing through point (1,1). Since

$\tan(\theta)=\frac{y}{x}$ we have $\tan(\theta)=\frac{1}{1}$ or $\theta=\tan^{-1}(1)$ so $\theta=\frac{\pi}{4}$

However, if the ray instead passes through (-1,-1), we get the same result since (-1/-1) is also equal to 1, but the angle is actually $(-3\pi)/4$. This difficulty was addressed by the designers of the Fortran programming language, which includes a function called *atan2(y,x)* to find the arc tangent of y/x correctly in any of the four quadrants, by accounting for the signs of *x* and *y*.

It is a simple matter to accomplish this in TI Basic with the built-in *R▸Pθ()* function, if we account for the special case (0,0). In fact, it can be done with a single line of TI Basic code:

```
when(x=Ø and y=Ø,undef,R▸Pθ(x,y))
```

*undef* is returned for (0,0), otherwise *R▸Pθ(0,0)* returns itself in Radian angle mode, and this expression in Degree angle mode:

```
18Ø*R▸Pθ(Ø,Ø)/π
```

Neither of these results are useful.

A more elaborate function can be written which also handles list and matrix arguments:

```
atan2(αx,αy)
Func
©(x,y) 4-quadrant arctan(y/x)
©Must be installed in math\
©6janØ2/dburkett@infinet.com

local αt,εm,τx                               © Function name, error message, αx type

"atan2 error"→εm                             © Initialize error message

define αt(α,β)=func                          © Function finds atan2() of simple elements
 when(α=Ø and β=Ø,undef,R▸Pθ(α,β))
endfunc

getType(αx)→τx                               © Save argument type for later tests

if τx≠getType(αy):return εm                  © Return error if arguments not same type

if τx="LIST" then                            © Handle list arguments
 if dim(αx)≠dim(αy):return εm
 return seq(αt(αx[k],αy[k]),k,1,dim(αx))

elseif τx="MAT" then                         © Handle matrix arguments
 if rowdim(αx)≠rowdim(αy) or coldim(αx)≠coldim(αy):return εm        © Validate dimensions
 return list▸mat(math\atan2(mat▸list(αx),mat▸list(αy)),coldim(αx))

elseif τx="NUM" then                         © Handle numeric arguments
 return αt(αx,αy)

else                                         © Return error for all other arg types
 return εm

endif

EndFunc
```

Both arguments of *atan2()* must be the same type, and must be numbers, lists or expressions. *atan2()* does not work with symbolic arguments.

Typical calls and results in Degree angle mode are:

| | | |
|---|---|---|
| atan2(1,1) | returns | 45 |
| atan2(-1,-1) | returns | -135 |
| atan2({1,-1},{1,-1}) | returns | {45,-135} |

$$a\tan 2\left(\begin{bmatrix} 1 & -1 \\ 0 & 3 \end{bmatrix}, \begin{bmatrix} 1 & 1 \\ 0 & \sqrt{3} \end{bmatrix}\right) \qquad \text{returns} \qquad \begin{bmatrix} 45 & 135 \\ \text{undef} & 30 \end{bmatrix}$$

### [6.49] Taylor() polynomial function finds tangent line

The line tangent to a function y = f(x) at a point x = a is defined by the criteria that the line passes through point (a,f(a)) and the slope of the line is the derivative of f(x) at x = a. If the tangent line is

$$y = m \cdot x + n \qquad [1]$$

then $$m = \frac{d}{dx} f(x) \mid x = a \qquad [2]$$

and if f(a) = b, we find n with $$n = b - m \cdot a \qquad [3]$$

The tangent line is the first-order Taylor series polynomial at x = a, so we can use the built-in *taylor()* function to find the line equation. The syntax for *taylor()* is

```
taylor(expr, var, order [,point])
```

where *expr* is the expression, *var* is the indpendent variable, *order* is the polynomial order, and the optional *point* argument is the point at which to find the series expansion. To find the tangent line, we set *order* = 1 and *point* = a. For example, if

$$y = 2 \cdot x^2 - 7 \cdot x + 1$$

and we want to find the tangent line at x = 3, then use

```
taylor(2*x^2-7*x+1,x,1,3)
```

which returns 5(x - 3) - 2. Use *expand()* on this result to put it in the more common form of 5x - 17.

We can also use this method to find symbolic results. For example, suppose

$$y = \ln\left(x^2\right)$$

and we want to find the tangent line at x = a, then use

```
expand(taylor(ln(x^2),x,1,a))
```

which returns $\frac{2 \cdot x}{a} + \ln\left(a^2\right) - 2$

This method fails if the CAS cannot find the Taylor polynomial for the expression. This is the case for complicated user-defined functions. In this case, you can find the derivative numerically (see, for example, tip 6.26), then solve for the tangent line constant with equation [3] above.

### [6.50] The Savage Benchmark

In March 2001 a discussion on the comp.sys.hp48 news group examined the Savage benchmark and the performance of various HP and TI calculators with that benchmark. In this tip I describe that benchmark, and summarize some test results provided by HP and TI users. More importantly, I offer some opinion as to why benchmarks are fairly silly, particularly one as simple-minded as the Savage benchmark.

A benchmark is a program intended to measure some attribute of a computer. The attribute may be accuracy, execution time, code compactness or efficiency, or cost. Most generally, benchmark users want to compare the performance of two or more computers, with the net result being an evaluation as to which is the 'best' computer in terms of what the benchmark measures.

For perspective, Eric S. Raymond offers this definition in *The Jargon Lexicon*

> *benchmark n.*
>
> *[techspeak] An inaccurate measure of computer performance. "In the computer industry, there are three kinds of lies: lies, damn lies, and benchmarks." Well-known ones include Whetstone, Dhrystone, Rhealstone (see h), the Gabriel LISP benchmarks (see gabriel), the SPECmark suite, and LINPACK. See also machoflops, MIPS, smoke and mirrors.*
>
> *(http://www.tuxedo.org/~esr/jargon/html/entry/benchmark.html)*

There are at least two fundamental objections to benchmarks with regards to applying them to calculators. The first is that a simple benchmark by its very nature measures a narrow aspect of the calculator's performance. The second objection is not the fault of the benchmark, but the benchmark results can be taken well out of context to infer that one calculator is 'better' than another in some vague, broad sense.

I do not claim that some HP calculator is 'better' than some TI calculator, or vice versa, because such a claim cannot be made on the basis of a single benchmark. Instead, I examine this benchmark because it is instructive and interesting as a specific example of a benchmark.

The Savage benchmark is a simple iterative floating-point calculation:

1. Set a = 1
2. Set a = tan(atan(exp(ln(sqrt(a*a)))))+1
3. Repeat step 2 *n* times

where

tan() is the tangent function
atan() is the arc-tangent function
exp() is the natural-logarithm-base *e* exponentiation function *e*^x
ln() is the natural logarithm function
sqrt() is the square root function
n is the number of loop iterations

and the calculation is performed with the angle mode in radians.

The two results for this benchmark are relative accuracy and execution time. The relative accuracy is (a-(n+1))/(n+1), where *a* is the final value of *a*. Note that relative accuracy can be interpreted as the number of correct significant digits. If the relative accuracy is 1E-9, then we have nearly nine accurate significant digits. In general, the execution time is specified as time/n, or the time for each iteration. If the same number of iterations is used for both calculators, then the total execution time can be reported and compared.

If all the functions in the benchmark calculation were performed with no loss of precision, the relative accuracy would be zero, since *a* reduces to n+1.

Benchmarks can be grouped in two classes, 'synthetic' and 'live'. A live benchmark attempts to simulate a realistic mix of the all the operations the user would perform with the calculator. A synthetic benchmark attempts to measure a much more narrow aspect of performance. The Savage benchmark is clearly a synthetic benchmark, because it only measures a few functions out of the hundreds available on the calculator. Further, it is particularly 'synthetic' because we would never go to the trouble to actually calculate this expression as a real problem: we know that the final result is n+1.

For some reason, we have decided that n =  2499 is appropriate for benchmarking our calculators. There is nothing magic about this number. As I will show later, the relative accuracy is strongly dependent on *n*, even over a relatively narrow range. This makes the relative accuracy after *n* iterations a poor measure of performance. Faster computers typically use n = 100,000 or 250,000. If the computer (or calculator) does not have built-in timer facilities, then the timing must be done by hand. In this case, a large value for n reduces relative uncertainty from human reaction time in starting and stopping the stopwatch.

This table shows the relative error and execution time for a variety of calculators and languages. I cannot vouch for the accuracy, since most results were culled from posts on the HP and TI discussion groups.

| **Model** | **Language** | **Execution time, sec** | **Relative accuracy** | **Reported by:** |
|---|---|---|---|---|
| HP-15C | | ~45 min | n/a | Mike Morrow |
| HP-20S | | 369 | -2.054E-7 | Mike Morrow |
| HP-28S | | 255 | -2.054E-7 | Mike Morrow |
| HP-32Sii | | 451 | -2.054E-7 | Mike Morrow |
| HP-41CX | | ~45 min | n/a | Mike Morrow |
| HP-42S | | 602 | -2.054E-7 | Mike Morrow |
| HP-48SX | UserRPL? | 199 | -2.054E-7 | Mike Morrow |
| HP-48GX | UserRPL? | 112 | -2.054E-7 | Mike Morrow |
| HP-48GX | Saturn ASM, display on, 15-digit | 70 | -3.572E-9 | Jonathon Busby |
| HP-48GX | Saturn ASM, display off, 15-digit | 62 | -3.572E-9 | Jonathon Busby |
| HP-49G | UserRPL | 120 | -2.054E-7 | Ralf Fritzsch |
| HP-49G | SysRPL, 12-digit, Display on | 96 | -2.054E-7 | Ralf Fritzsch |
| HP-49G | SysRPL, 12-digit, Display off | 87 | -2.054E-7 | Thomas Rast |
| HP-49G | SysRPL, 15-digit | 138 | -3.572E-9 | Ralf Fritzsch |
| Psion XPII | OPL Basic | 47 min | -1.022E-5 | Doug Burkett |
| TI-85 | | 368 | -3.08E-9 | Mike Morrow |
| TI-86 | TI Basic | 433 | -3.081E-9 | Ralf Fritzsch |
| TI-86 | Z80 assembler | 328 | -3.081E-9 | Ralf Fritzsch |
| TI-89 HW1 | C (GCC compiler), AMS 2.05 | 138 | 1.011E-9 | Doug Burkett |
| TI-89 HW1 | TI Basic, AMS 2.05 | 272 | -3.081E-9 | Doug Burkett |
| TI-92+ | TI Basic (HW1? AMS?) | 255 | -3.081E-9 | Jacek Marchel |
| TI-92+ | 68K assembler | 72 | -3.081E-9 | Jacek Marchel |
| TI-92+ HW2 | C (GCC compiler) AMS 2.05 | 96 | 1.011E-9 | Doug Burkett |
| TI-92+ HW2 | TI Basic, AMS 2.05 | 187 | -3.081E-9 | Doug Burkett |

The best relative accuracy is returned with the TI-89 / TI-92 Plus, with the GCC C compiler program. The best execution time is returned by the HP-48GX with an ASM program and the display turned off.

So what can we conclude from these results? Very little, actually. We can conclude, for this very synthetic problem, that the relative accuracies range from less than 7 to nearly 9 significant digits. We see that newer hardware is more accurate and has faster execution times. We see that the choice of programming language has a strong effect on the execution time. None of this is news; there are no surprises here. Instead, what cannot be concluded is more significant:

- We cannot conclude that the HP-48GX is, in general, faster at numeric computation, because we have only tested a tiny subset of all the functions.
- We cannot conclude that the TI-89 / TI-92 Plus is, in general, more accurate, because again, only a tiny subset of the functions has been tested.

- We cannot even conclude that one calculator is more accurate than any other, even for these functions, because we have not actually tested function evaluation accuracy. Instead, we have tested the combined accuracy of three functions (tangent, e^x and square root) and their inverses. It is possible (though unlikely) that errors in one function evaluation are compensated by opposite errors in the inverse function.

To use the HP-49G and the TI-92 Plus as an example, all we have shown is that either calculator may or may not be faster and more accurate than the other.

There is much data missing from the table above, and this also precludes a fair comparison. We don't know, in each case, which versions of operating systems and compilers were used. We don't know which hardware versions were used. Running changes in hardware and software design may mean that clock rates and algorithms may have been changed. For a truly meaningful benchmark, all these variables must be specified.

In regards to the execution time, we have not really measured the function evaluation time, anyway. Instead, we have measured the combined execution time of the function evaluation and the loop overhead. For the TI-89 / TI-92 Plus TI Basic programs, this distinction is significant because the loop overhead is significant: 46 seconds for the TI-89 HW1, and 31 seconds for the TI-92 Plus HW2. This amounts to about 18 mS/iteration for the TI-89, and 12 mS/iteration for the TI-92 Plus.

This is the TI Basic code used for the TI-89 / TI-92 Plus benchmarks:

```
savage2()
Func
local a,n
1→a
for n,1,2499
 tan(tan⁻¹(ℯ^(ln(√(a*a)))))+1→a
endfor
(a-2500)/2500
EndFunc
```

This is the GCC C code used for the TI-89/92+ benchmarks:

```
// C Source File
// Created 3/22/2001; 12:21:38 PM

#define RETURN_VALUE      // Redirect Return Value

// #define OPTIMIZE_ROM_CALLS // Use ROM Call Optimization

#define SAVE_SCREEN           // Save/Restore LCD Contents

#include <tigcclib.h>         // Include All Header Files

short _ti89;                  // Produce .89Z File
short _ti92plus;              // Produce .9XZ File


// Main Function
void _main(void)
{
float a=1.0;
int k;
a=1.0;
for(k=1;k<2500;k++) a=tan(atan(exp(log(sqrt(a*a)))))+1.0;
push_Float ((a-2500.0)/2500.0);
}
```

The C version was written by Artraze (http://www.angelfire.com/pa3/cartethus/).

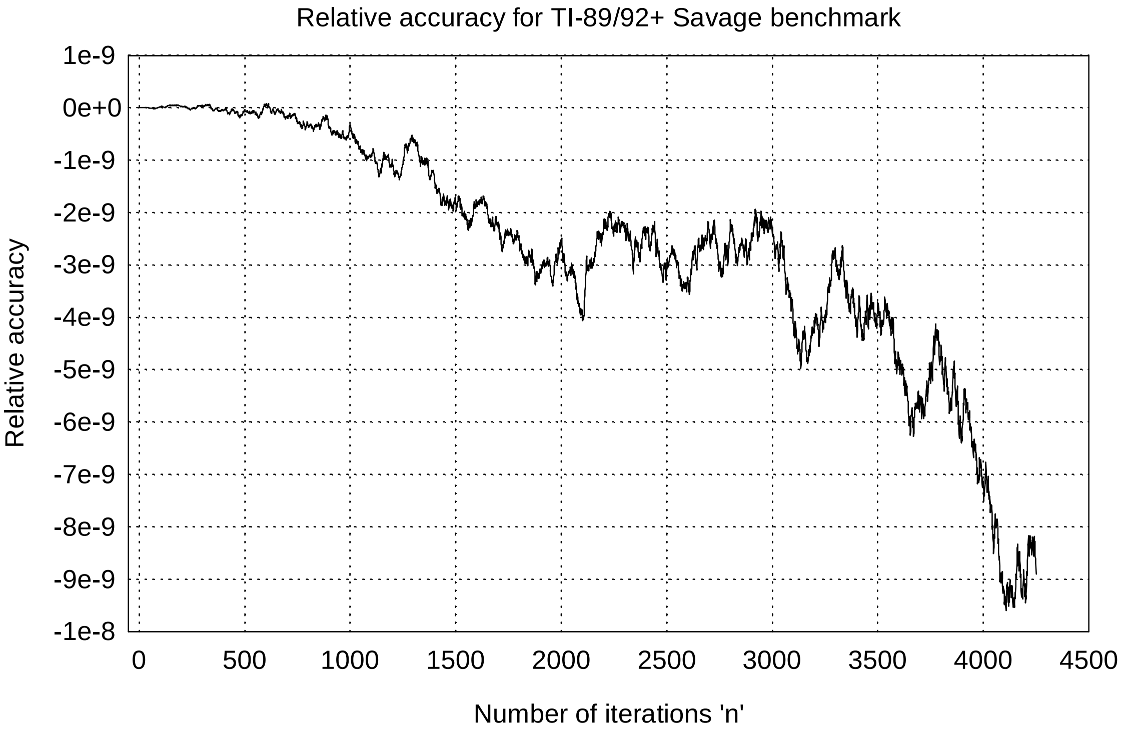
Note that the C version is 641 bytes, and the TI Basic version is only 83 bytes.

There has been some confusion about the number of significant digits used by the TI-89 / TI-92 Plus, which can be cleared up by referring to the TI SDK manual. In usual calculator operation, 12 significant digits are displayed, except for regression results, in which 14 significant digits can be shown. 14 significant digits are used for 'normal' calculator operation, and for TI Basic calculations. However, C programs can use the C 'double' floating point, type, as this quote from section 16.4 describes:

> *"Applications can work with float numbers on the estack or in C floating-point variables. The compiler supports two forms of floating-point values as described in Chapter 2 of the compiler documentation. The calculator implementation uses the standard C type double. The symbols BCD16 and Float are also defined to be double. BCD16 is the recommended type for declaring floating-point variables in applications.*
>
> *"This type uses a 16-digit mantissa and provides more accuracy. Thus, BCD16 variables provide the best results when implementing iterative algorithms that require a great deal of floating-point computation.*
>
> *"push_Float is the routine that converts a C floating-point value into a tagged floating-point value on the expression stack. The 16-digit value is rounded to 14-digits, pushed onto the estack, and then a FLOAT_TAG is pushed on top."*

The C program listed above defines *a* as double, and this is why the TI-89/92+ relative accuracy is slightly better than that of the HP-49G. For the vast majority of applications, this difference is insignificant.

I mentioned earlier that the relative accuracy at a single point is a poor indicator of accuracy. The plot below shows the relative accuracy at each calculator point, up to 4,250 points. This data is from the TI-89 / TI-92 Plus TI Basic program.



As expected, the relative error starts out at zero, then increases with the number of iterations. However, the error does not increase smoothly, and varies considerably over short ranges. For example, the relative error is about -3E-9 at 2500 iterations, but was -2E-9 at about 2200 iterations. Since the actual shape of the relative error curve will depend on the number of significant digits, as well as the algorithms used for the evaluated functions, some other statistical measure would be a better indication of the relative accuracy. Some candidates are the RMS error over some range, or even the mode. Of course, the best description of the relative error is given by the error plot itself.

References:

*Byte* magazine, 1985, vol 10 no 11
*Sky & Telescope* magazine, March 1987

Neither of these references discuss the benchmark in much detail.

### [6.51] Review of TI-89 / TI-92 Plus Calculus Tools

*(This review was written by George C. Dorner, September, 2001)*

This is a nice little package of programs for these two TI calculators. The package is downloadable for either calculator (which must be using the latest version of the calculator operating system, V. 2.03) from the TI online store at http://epsstore.ti.com/. Here's one of the best parts: it's free!

The package is a collection of programs which will be useful to students and teachers, mostly in calculus classes or courses where calculus is used. Once the package is loaded, the programs and functions basically become part of the calculator firmware.

Here are the programs (functions) and a comment about each:

*Tangent Line and Normal Line:*
These two functions compute the equations of the respective lines, given a function and a value of the independent variable. Each then displays a graph of the function and the respective line. This will be useful to teachers while teaching the derivative concept and to students to check their understanding and their homework.

*Newton's Method:*
For a given function and an initial guess of the independent variable, the graph of the function and successive tangent lines which estimate a zero of the function are drawn. This is nice to illustrate the Method and possibly to find a zero of a function, though there are better methods built in to the calculator. I use this to introduce Newton's Method, though.

*Implicit Derivative:*
This finds dy/dx given f(x,y)=0. Useful to students, but most teachers will prefer to use a method which shows the steps and concepts involved. The practical use of this function is that it will find nth derivatives implicitly, a topic which is often slighted and rarely mastered in the first course.

*Curvature, Center of Curvature, and Osculating Circle:*
These do the computations for any function and will be useful to teacher and student while studying these topics. The answers are given "for any x", not just for specified values of the independent variable.Thus, it is easy to explore the local behavior of a curve in 2-space and related topics of geometry such as the evolute.

The section on integration will be very useful at the outset of learning about the definite integral. It contains functions to display the graph of a chosen function on a chosen interval, together with a selected number of "approximating panels" using any of the midpoint, left-hand, right-hand, trapezoidal, or Simpson's Rule. Also, there is an option to display the Riemann sums for all methods on the same screen. Armed with the right examples, there is plenty here to clarify the teaching and understanding of the limit of such sums.

A function is included which performs the ratio test for infinite series, given a formula for the general term. Two other functions included here are not traditional calculus topics., They find the sum of either first order or second order difference equations, given the expression of the general term. In a course where discrete methods or dynamical systems are explored, these could be very useful.

Computation of four functions, the gradient, divergence, curl, and potential function, which are studied in vector calculus is included. As is the case with all the functions of this package, these may be called during the regular use of the calculator without previously entering the Calculus Tools menu. This substantially extends the usefulness of the calculator.

There are eight "Advanced" mathematical functions included. The error function, *erf()*, and the complementary error function, *erfc()*, will be useful in statistics applications, but these are also included in another free downloadable FlashWare application which duplicates the statistics functions on the TI-83. In fact, the function *TIStat.normCdf()* seems to be somewhat more accurate than *CalcTool.erf()*.

The gamma function is included here, too. Its presence may encourage exploration of this complicated topic during the second calculus course and will be an occasional aid to students in applied courses. Its accuracy seems to be good beyond 5D, plenty for beginners.

“Automatic” computation of the Fourier series of a periodic function is provided by another useful function in this utility package. It will compute a designated number of terms in the Fourier series of a function specified on a period interval. A graph of the partial sum is displayed. This will be very useful to those who wish to bring this important applied topic to a calculus course audience.

Five integral applications round out the Advanced portion. The first allows the evaluation of a two-fold iterated integral over a region which is described “curve-to-curve” and “point-to-point”, what are called Type I or Type II regions in some books. More generally, it allows the surface integral of a function over a surface given in the form z=f(x,y), weighted by a “density function”, d(x,y). The terminology and names used in these applications are not standard to most calculus books, but they will be somewhat familiar to students of physics and mechanics. More general terminology and more examples would make this more useful to students, but the function is very powerful. Another “integral of Density” for a polar region is provided by a second function here. It simply evaluates the integral of a function of r and theta over a “nice” region defined in polar coordinates.

The “Centroid of Density” function returns the coordinates of the centroid of a plane region in rectangular coordinates which is weighted with a density function depending on the two rectangular coordinate variables. And, for the same data, the “Inertia Tensor” function returns a 2X2 matrix with entries which are the moments of inertia with respect to the two axes and the (equal) products of inertia. This terminology is straight from a mechanics or physics book and doesn’t appear in most calculus books., where moments of inertia _are_ discussed. (I wonder how many calculus students will _ever_ know what an inertia tensor is?)

The last function is the curiously named “Arc Displacement” , which most calculus texts refer to as “arc length in polar coordinates.” The function name is ‘plrArcLn( ), whch seems appropriate. Again, there is a provision for including a weight function to “distribute” mass or charge or some other quantity along the curve. Thus, a problem of finding the total mass of a non-uniform wire in the shape of a known curve is facilitated.

There is a nice 40 page Acrobat manual describing the Calculus Tools package. In places it is a little too terse for my taste. The formulas which are the basis of the functions provided are not typically included, leaving open their exact interpretation. I ended up doing the examples by hand to convince myself of their meaning, but some users may be buffaloed by terminology and a paucity of examples.

My complaints are thus more about the clarity of what is available than of the execution or utility of the mathematics here. More standard terminology and a few more examples would improve this neat addition to the TI FlashWare arsenal. This will be useful on occasion to most students and users of calculus, and it could be very useful during a learning process involving any one of the techniques or topics covered. I hope we see more such packages. This one was produced by a cooperative effort of a number of the gurus of these calculators, and not by TI headquarters. Most are not mathematics teachers. I hope that this is a precursor of similar packages to come from other users incorporating material which they have found genuinely useful to themselves and not just included for commercial impact.

### [6.52] Return complex result in REAL mode with 0i

In general, complex results are not returned if the Complex Format mode is set to REAL. For example,

```
ln(-e^(x))
```

gives the *Non-real result* error. However, the correct result is returned if you include the complex *i*:

```
ln(-e^(x))+Øi          returns          x+i*π
```

Note that adding 0*i* to the original expression does not change it.

*(Credit to Kevin Kofler)*

## [6.53] Implicit derivatives

An implicit derivative results from differentiating an implicit function of the form f(x,y) = 0. Implicit differentiation is covered in all basic calculus texts, so I will skip the details here, and focus on finding implicit derivatives with the TI-89/TI-92+.

The most straight-forward method to find the implicit derivative of f(x,y) is to specify y = y(x) in the expression and use the built-in differentiation function *d*(). For example, to find the implicit derivative of

$$f(x,y)=4\cdot x^2-6\cdot x\cdot y+3\cdot y^2$$

use `d(4*x^2-6*x*y(x)+3*(y(x))^2,x)`

or `d(4*x^2-6*x*y+3*y^2,x)|y=y(x)`

which both return $(6\cdot y(x)-6\cdot x)\cdot\frac{d}{dx}(y(x))-6\cdot y(x)+8\cdot x$

Note that the derivative occurs as a single term, so you can solve for it. First, copy the expression from the history area, and substitute for the derivative; I use the temporary variable *yp*:

```
(6*y(x)-6*x)*d(y(x),x)-6*y(x)+8*x|d(y(x),x)=yp
```

which returns `(6*yp-6)*y(x)+(8-6*yp)*x`

Now set the expression equal to zero and solve for the derivative:

```
solve((6*yp-6)*y(x)+(8-6*yp)*x=Ø,yp)
```

which returns $yp=\frac{3\cdot y(x)-4\cdot x}{3\cdot(y(x)-x)}$

You can repeat this process to find higher-order derivatives.

If you need to solve many implicit derivatives, particularly of higher orders, it is worthwhile to define a function. The following routine is a modified version of routines by David Stoutemyer and Don Phillips.

```
impdifn(ü,x,y,ñ)
Func
Local  e,e1,i

If  part(ü,Ø)="="            © Convert equation to expression if necessary
left(ü)-right(ü)→ü

ˉd(ü,x)/(d(ü,y))→e1          © Find first derivative
e1→e
```

```
For  i,2,ñ                    © Loop to find higher-order derivatives
 d(e,x)+e1*d(e,y)→e
EndFor

e                             © Return derivative.
                              © (Replace with factor(e) to automatically factor result.)
EndFunc
```

The arguments are

```
impdifn(expression, independent variable, dependent variable, order)
```

*expression* may be an algebraic expression, or an equation. *order* is the order of the derivative: 1 for the first derivative, 2 for the second and so on.

For example, the call to find the second derivative of x^4 + y^4 = 0 is

```
impdifn(x^4+y^4,x,y,2)
```

which returns $$\frac{-3\cdot x^{6}}{y^{7}}-\frac{3\cdot x^{2}}{y^{3}}$$

You can apply *factor()* to this expression to get $$\frac{-3\cdot x^{2}\cdot\left(x^{4}+y^{4}\right)}{y^{7}}$$

which is zero, since x^4 + y^4 = 0.

Don Phillips modified David Stoutemyer's *impdifn()* to handle equations as input arguments, and automatically factor the result. I modified Don's routine by removing the automatic factoring of the result (you can do this manually, when needed), and converting a *Loop ... EndLoop* to a *For* loop.

If you use the first, manual method to find the derivative of an equation, this warning may be shown in the status line: *Warning: Differentiating an equation may produce a false equation*.

*(Credit to Bhuvanesh Bhatt and Kevin Kofler)*

### [6.54] Delete variables after using numeric solver

The variables in the numeric solver exist in the current folder after exiting the solver, which can be useful if you run the solver again with the same equation. Since the variables do take up memory and may interfere with further calculations, this tip shows a convenient method to delete them.

The current solver equation is stored in the system variable *eqn*. We can use the RCL feature to recall *eqn* without variable substitution, so we can see the variables. Then, the functions *delvar1()* and *exprvars()* can automatically extract and delete the variables. *exprvars()* returns the *eqn* variables as a list with string elements, and *delvar1()* deletes a list of variables. Assuming that *delvar1()* and *exprvars()* are both stored in the *util* folder, enter this:

```
util\delvar1(util\exprvars("
```

Next press [RCL] to display the Recall Variable dialog box. [RCL] is [2nd] [STO] on both calculators. Type [e] [q] [n] [ENTER] [ENTER] to recall the *eqn* system variable, then type ["] [ ) ] [ ) ] [ENTER]. The variables are then deleted.

For example, suppose the current equation is a=b+c, then the complete entry line looks like this before the final [ENTER] is pressed:

```
util\delvar1(util\exprvars("a=b+c"))
```

If *eqn* is an expression instead of an equation, then *eqn* has the form *exp* = *expression*, where *exp* is a system variable. This method will delete *exp* along with the other variables, since *exprvars()* includes *exp* in the variable list.

The method can be automated with this program:

```
deleqnv()
Prgm
©Delete vars in eqn
©28dec01/dburkett@infinet.com
local ö,ü

Try:newFold(ä):else:endTry          © Create new folder if necessary
setFold(ä)→ö                        © Make new folder current & save old folder
string(eqn)→ü                       © Convert eqn to string, without var substitution
setFold(#ö)                         © Restore old folder

util\delvar1(util\exprvars(ü))      © Extract eqn variables & delete them

EndPrgm
```

*deleqnv()* calls *delvar1()* and *exprvars()*, both of which must be in the *util*\ folder. The bulk of the program recovers *eqn* as a string without variable substitution. For more details on this method, see tip [7.40], *Recall expression without variable value substitution*. *delvar1()* is described in tip [7.39], *Quickly delete locked, archived variables*. *exprvars()* is described in tip [7.42], *Find variable names used in expressions*.

This program will not work if you change folders after using the numeric solver.

### [6.55] Algorithms for factor() and isPrime()

The following description of the algorithms for *factor()* and *isPrime()* was obtained from TI by Bhuvanesh Bhatt. It is included with TI's permission.

"The algorithms used in the TI-92, TI-92 Plus, and TI-89 are described in D. Knuth, The Art of Computer Programming, Vol 2, Addison-Wesley, but here is a brief description:

"The TI-92 simply divides by successive primes through the largest one less than 2^16.1: It doesn't actually keep a table or use a sieve to create these divisors, but cyclically adds the sequence of increments 2, 2, 4, 2, 4, 2, 4, 6, 2, 6 to generate these primes plus a few extra harmless composites.

"TI-92 Plus and TI-89 start the same way, except that they stop this trial division after trial divisor 1021, then switch to a relatively fast Monte-Carlo test that determines whether the number is certainly composite or is almost certainly prime. (Even knowing the algorithm, I cannot construct a composite number that it would identify as almost certainly prime, and I believe that even centuries of continuous experiments with random inputs wouldn't produce such a number).

"The *isPrime()* function stops at this point returning either false or (almost certainly) true.

"If the number is identified as composite, the *factor()* function then proceeds to the Pollard Rho factorization algorithm. It is arguably the fastest algorithm for when the second largest prime factor has less than about 10 digits. There are faster algorithms for when this factor has more digits, but all known algorithms take expected time that grows exponentially with the number of digits. For example, the Pollard Rho algorithm would probably take centuries on any current computer to factor the product of two 50-digit primes. In contrast, the *isPrime()* function would quickly identify the number as composite.

"Both the compositivity test and the Pollard Rho algorithm need to square numbers as large as their inputs, so they quit with a *"Warning: ... Simplification might be incomplete."* if their inputs exceed about 307 digits."

## [6.56] Fourth-order splice joins two functions

A fourth-order splice joins two functions smoothly over an interval. The splice function matches the two functions at the interval boundaries, and the first derivatives of the splice equal those of the two functions. The fourth-order splice uses a point in the interval interior which controls the splice behavior. This splice is useful to join two functions, such as regression models, resulting in a model (with three equations) which is continuous and smooth, in the first derivative sense, over the interval of interest.

This tip is organized in these sections:

*Splice function derivation*
*The function splice4()*
*Example: approximate the sin() function*
*Differentiating the splice*
*Integrating the splice*
*Solving for the splice inverse*
*User interface program for splice4()*
*Scaling the derivatives*
*Scaling the splice integral*

These programs and functions are developed and described:

| | |
|---|---|
| *splice4()* | Calculate the splice function coefficients |
| *spli4ui()* | User interface for *splice4()* |
| *spli4de()* | Calculate the splice derivative |
| *spli4in()* | Calculate a numeric derivative for the splice |
| *spli4x()* | Calculate x, given the splice value of y. Uses *nsolve()* |
| *spli4inv()* | Find an approximate polynomial for the inverse of the splice function |

### *Splice function derivation*

Define

$x_1$ = the left interval bound
$x_3$ = the right interval bound
$x_2$ = the interval interior control point, where $x_1 < x_2 < x_3$
$h = x_2 - x_1 = x_3 - x_2$ (the splice half-width; the control point is midway between the interval bounds)
$f_1(x)$ is the left-hand function to be spliced
$f_2(x)$ is the right-hand function to be spliced

s(x) is the splice function

In general, the splice function is $s(x) = a \cdot x^4 + b \cdot x^3 + c \cdot x^2 + d \cdot x + e$ [1]

and the first derivative is $s'(x) = 4 \cdot a \cdot x^3 + 3 \cdot b \cdot x^2 + 2 \cdot c \cdot x + d$ [2]

$x_2$ is the point between $x_1$ and $x_3$ where we specify some desired value for the splice function s(x). There are several choices for $s(x_2)$. For example, if $f_1(x)$ and $f_2(x)$ intersect, we may want to set $x_2$ as the point of intersection, and set $s(x_2) = f_1(x_2) = f_2(x_2)$. Alternatively, the splice may work better by choosing a value for $x_2$ slightly away from the intersection. If the functions do not intersect on the splice interval, we may set $s(x_2)$ between the functions so the splice passes smoothly from one to the other.

We solve for the five coefficients of the splice polynomial by imposing five conditions:

1. $f_1(x_1) = s(x_1)$ The splice matches $f_1$ at $x_1$; there is no discontinuity
2. $f_2(x_3) = s(x_3)$ The splice matches $f_2$ at $x_3$; there is no discontinuity
3. $s(x_2) = y_2$ We set the splice value at $x_2$ to get some desired behavior
4. $\frac{d}{dx} f_1(x_1) = \frac{d}{dx} s(x_1)$ Set the first derivatives equal at $x_1$ for a smooth transition
5. $\frac{d}{dx} f_2(x_3) = \frac{d}{dx} s(x_3)$ Set the first derivatives equal at $x_3$ for a smooth transition

so we solve these five equations for a, b, c, d and e:

$a \cdot x_1^4 + b \cdot x_1^3 + c \cdot x_1^2 + d \cdot x_1 + e = f_1(x_1)$ (condition 1) [3]

$a \cdot x_3^4 + b \cdot x_3^3 + c \cdot x_3^2 + d \cdot x_3 + e = f_2(x_3)$ (condition 2) [4]

$a \cdot x_2^4 + b \cdot x_2^3 + c \cdot x_2^2 + d \cdot x_2 + e = y_2$ (condition 3) [5]

$4 \cdot a \cdot x_1^3 + 3 \cdot b \cdot x_1^2 + 2 \cdot c \cdot x_1 + d = f'_1(x_1)$ (condition 4) [6]

$4 \cdot a \cdot x_3^3 + 3 \cdot b \cdot x_3^2 + 2 \cdot c \cdot x_3 + d = f'_2(x_3)$ (condition 5) [7]

This set of five equations in five unknowns can sometimes be solved, but just as often they cannot. As we typically fit the splice over a narrow range of *x*, a *singular matrix* error often results from loss of precision in solving for the coefficients. Even if the *singular matrix* error does not occur, the solved coefficients are large and alternating in sign, which means that the polynomial is numerically unstable. This problem is avoided by finding the splice polynomial as a function of a scaled value of $x_s$, instead of *x* itself. As usual, proper scaling simplifies the solution. For example, suppose we set $x_{1s} = -1$, $x_{2s} = 0$ and $x_{3s} = 1$, where $x_{1s}$, $x_{2s}$ and $x_{3s}$ are the scaled values of $x_1$, $x_0$ and $x_3$, respectively. The splice polynomial is now $s(x_s)$:

$$s(x_S) = a \cdot x_S^4 + b \cdot x_S^3 + c \cdot x_S^2 + d \cdot x_S + e \quad [9]$$

Above, we defined $h = x_2 - x_1$; now we define a scaled interval half-width $h_s$:

$$h_s = x_{2s} - x_{1s} = x_{3s} - x_{2s} = 1 \qquad [10]$$

We also define a scaling factor *k* such that

$$k = \frac{h_s}{h} = \frac{1}{h} \qquad [11]$$

Since we have scaled (transformed) the x- and y-data, we must also scale the derivatives used in [6] and [7]. It turns out that we can easily transform the derivatives as

$$f'_{1s}\left(x_{1s}\right) = \frac{1}{k} \cdot f'_1\left(x_1\right) \qquad [12]$$

$$f'_{2s}\left(x_{3s}\right) = \frac{1}{k} \cdot f'_2\left(x_3\right) \qquad [13]$$

These relations are derived in a section at the end of this tip. We can write [3] to [7] as a matrix equation

$$M \cdot c = v \qquad [14]$$

where *c* is the coefficient vector we need, and

$$M = \begin{bmatrix} x_{1s}^4 & x_{1s}^3 & x_{1s}^2 & x_{1s} & 1 \\ x_{2s}^4 & x_{2s}^3 & x_{2s}^2 & x_{2s} & 1 \\ x_{3s}^4 & x_{3s}^3 & x_{3s}^2 & x_{3s} & 1 \\ 4 \cdot x_{1s}^3 & 3 \cdot x_{2s}^2 & 2 \cdot x_{1s} & 1 & 0 \\ 4 \cdot x_{3s}^3 & 3 \cdot x_{3s}^2 & 2 \cdot x_{3s} & 1 & 0 \end{bmatrix} \qquad c = \begin{bmatrix} a \\ b \\ c \\ d \\ e \end{bmatrix} \qquad v = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y'_{1s} \\ y'_{3s} \end{bmatrix}$$

We solve [14] for *c* as $c = M^{-1} \cdot v$ [15]

$x_{1s}$, $x_{2s}$ and $x_{3s}$ are always -1, 0 and 1, so $M^{-1}$ is a constant matrix:

$$M^{-1} = \begin{bmatrix} -0.5 & 1 & -.5 & -0.25 & 0.25 \\ 0.25 & 0 & -0.25 & 0.25 & 0.25 \\ 1 & -2 & 1 & 0.25 & -0.25 \\ -0.75 & 0 & 0.75 & -0.25 & -0.25 \\ 0 & 1 & 0 & 0 & 0 \end{bmatrix} \qquad [16]$$

Since the scaled matrix inverse is independent of the functions and the splice points, we need not calculate it each time we find a splice: it is pre-computed and appears as a constant in the program. Because of the scaling we chose, the inverse matrix elements are simple and exact as shown. This means no round-off error can result from matrix inverse, which would happen with un-scaled *x*-values.

The *c* vector in [15] is the scaled coefficients; but we want the un-scaled result y = s(x). This is found with equation [9] where

$$x_s = \frac{\left(x - x_2\right)}{h} \qquad [17]$$

so s(x) can be calculated with the TI-89/92+ *polyEval()* function:

```
polyEval({a,b,c,d,e},k*(x-x2))
```

*Do not* expand the splice polynomial with the relationship shown in [17]! While this may seem to simplify the polynomial, it undoes the benefit of scaling, and the resulting polynomial is again unstable.

Ultimately we use a single function for f(x), which is implemented as a nested *when()* function:

```
when(x<x1,f1(x),when(x<x3,s(x),f2(x)))
```

or, in terms of the splice evaluation

```
when(x<x1,f1(x),when(x<x3,polyEval({a,b,c,d,e},k*(x-x2)),f2(x)))
```

There are several typical operations we want to perform with the splice, such as interpolation, differentiation, integration, and solving for the inverse, that is, solve y=s(x) for x, given y. Interpolation is a simple matter of scaling x to $x_s$ and evaluating the splice polynomial, as just shown. Following sections show how to do integration, differentiation, and solving for the inverse. I also include a user interface routine to find the splice coefficients, which eases entry of the input parameters and calculates the fit errors.

You can use the same basic ideas I have described to develop higher-order splices. For example, you could use a 6th-order splice which forces the second derivatives to match at the interval endpoints, or an 8th-order splice that forces matching second- and third-order derivatives at the endpoints. You could change the 4th-order splice to a fifth-order splice which specifies a first derivative at the interval midpoint *x2*.

### *The function splice4()*

The function shown below, *splice4()*, implements the technique described above. The arguments are defined in the function header comments

*splice4()* returns the splice polynomial coefficients as a list {a,b,c,d,e}, which can be used directly with *polyEval()* to find s(x). The listing below includes comments which are not in the *splice4.9xf* file in *tlcode.zip*.

```
splice4(y_1,y_2,y_3,yp1,yp3)
Func
©(y1,y2,y3,y1',y3') 4th-order splice
©27jan02/dburkett@infinet.com

© Input arguments

© f1(x) and f2(x) are the two functions to be spliced between x1 and x3.

© y_1   f1(x1)
© y_2   s(x2)
© y_3   f2(x3)
© yp1   f1'(x1), scaled such that f1'(x1) = h*f1'(x1)
© yp3   f2'(x3), scaled such that f2'(x3) = h*f2'(x3)

© Output:
© Returns {a,b,c,d,e} where s(xs) = a*(xs)^4 + b*(xs)^3 + c*(xs)^2 + d*(xs) + e

© Solve for scaled coefficients
© Find list 'c' of scaled coefficients {a, b, c, d, e} by solving
©
```

```
©              --    --
©              | y_1  |  (y1 = f1(x1))
©              | y_2  |  (y2)
© c = M^(-1) * | y_3  |  (y3) = f2(x3))
©              | yp1  |  (scaled value of f1'(x1))
©              | yp3  |  (scaled value of f2'(x3))
©              --    --
©
© M^(-1) has been pre-computed. The following expression is all one line.


mat▸list([⁻.5,1,⁻.5,⁻.25,.25;.25,Ø,⁻.25,.25,.25;1,⁻2,1,.25,⁻.25;⁻.75,Ø,.75,⁻.25,⁻.25;Ø,1,
Ø,Ø,Ø]*[y_1;y_2;y_3;yp1;yp3])

EndFunc
```

The following example uses *splice4()* to splice two approximating functions.

### *Example: approximate the sin() function*

Suppose we want to estimate sin(x) for 0 < x < 0.78 radians. We have found these estimating functions:

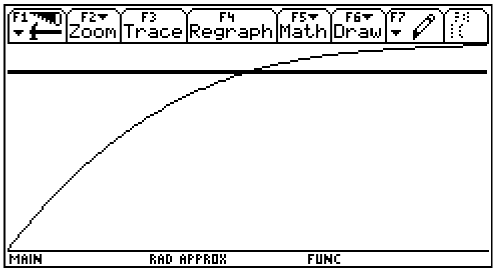
$$f_1(x) = k_1 \cdot x^3 + k_2 \cdot x^2 + k_3 \cdot x + k_4 \qquad \text{for x > 0 and x < ~0.546} \qquad [18]$$

$$k_1 = -0.15972286692682 \qquad k_3 = 1.0003712707863$$
$$k_2 = -0.00312600795332 \qquad k_4 = -4.74007298E-6$$

$$f_2(x) = k_5 \cdot x^2 + k_6 \cdot x + k_7 \qquad \text{for x > ~0.546 and x < 0.78} \qquad [19]$$

$$k_5 = -0.3073521499375 \qquad k_7 = -0.04107647476031$$
$$k_6 = 1.1940610623813$$

The first step is to set the center of the splice, $x_2$. We want to set the splice center near the boundary between $f_1(x)$ and $f_2(x)$, so as a starting point we plot the difference between the two estimating functions, which is $f_2(x)$ - $f_1(x)$. We plot the difference instead of the functions themselves, because both functions are so close to sin(x) that we would not be able to visually distinguish any difference. The plot below shows the difference over the range 0.45 < x < 0.65.



For this example the difference plot crosses the x-axis, so the functions intersect at this point. We choose this point as the splice center, so that the splice will not have to span a large distance between the two functions. Solving for the root gives

$$x_2 = 0.549220479094$$

In some cases, the two functions will not intersect at all, or at least in the desired splice range. In this case, a good choice for the splice center is the value of x at which the function difference is a minimum.

The next step is to set the width of the splice, which in turn sets the half-width *h*. Assuming the two estimating functions are both reasonably accurate, we want to make the splice rather narrow, but if we make it too narrow, accuracy degrades because of loss of significant digits when solving for the splice coefficients. For this example, I set h = 0.001, and we will verify that this is not too small. This means that the splice will be fit from about $x_1$ = 0.54822 to $x_3$ = 0.55022.

All that remains is to calculate the other *splice4()* arguments, with these steps.

In the Y= editor, define some functions to make the calculations (and plotting) easier. Note that the coefficients *k1* to *k6* have already been stored.

```
y1= polyEval({k1,k2,k3,k4},x)
y2= polyEval({k5,k6,k6},x)
y3= sin(x)
y4= y1(x)-y3(x)
y5= y2(x)-y3(x)
y6= polyEval(spl4,(x-x2)*k)
y7= y6(x)-y3(x)
```

*y1* and *y2* are $f_1(x)$ and $f_2(x)$. *y3* is the function to be estimated. *y4* and *y5* find the error between the model equations and the function to be estimated. *y6* will calculate the splice function, and *y7* will find the splice function error. Note that k = 1/h.

Next, enter these commands in the entry line:

```
.54922Ø48→x2
.ØØ1→h
1/h→k
y1(x2-h)→yy1
y1(x2)→yy2
y2(x2+h)→yy3
h*d(y1(x),x)|x=x2-h→yd1
h*d(y2(x),x)|x=x2+h→yd2
```

Finally, call the *splice4()* with the calculated arguments and save the result in *spl4*:

```
math\splice4(yy1,yy2,yy3,yd1,yd2)→spl4
```

Next we will ensure that the splice function meets the requirements. First, check the errors at $x_1$, $x_2$ and $x_3$. These expressions should all be near zero:

```
y1(x2-h)-y6(x2-h)      returns  2E-14
y1(x2)-y6(x2)          returns  Ø
y2(x2+h)-y6(x2+h)      returns  2E-14
```

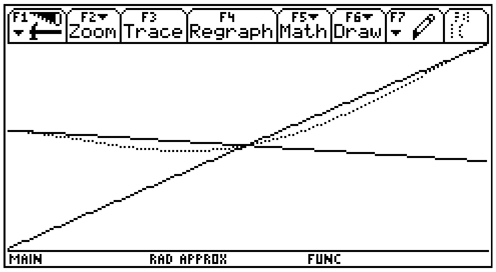
Next check the derivative errors at the splice endpoints. The derivative errors should be near zero. If we calculate the derivative errors (*incorrectly!*) by directly differentiating the splice function, we get

```
d(y1(x),x)-d(y6(x),x)|x=x2-h          returns 3.7629E-8
d(y2(x),x)-d(y6(x),x)|x=x2+h          returns  7.446E-9
```

The errors seem small enough, but calculating the derivatives correctly (as shown below) gives the true errors:

```
d(y1(xØ),xØ)-k*d(polyEval(spl4,x),x)|x=-1 and xØ=x2-h       returns       -6.Ø87E-11
d(y2(xØ),xØ)-k*d(polyEval(spl4,x),x)|x=1 and xØ=x2+h        returns       3.6Ø1E-11
```

So far, the splice seems to work well. As a final check, we graph the splice function over the splice interval. In the Y= editor, use [F4] to select *y4*, *y5* and *y7*. Because $f_1(x)$, $f_2(x)$ and the splice function are all so close together, plotting these *error* functions gives a better picture of the splice fit. To set the x-axis range, press [WINDOW] and set *xmin* to x2-h, and *xmax* to x2+h. Press [F2] [A] to plot the splice, which results in this graph:



The two solid traces are $f_1(x)$ and $f_2(x)$, relative to sin(x), and the dotted trace is the splice function, relative to sin(x). Note that the splice slope appears to match $f_1$ and $f_2$ at the endpoints, and it passes through $x_2$, as we specified. Based on the numerical check results, and the plotted result, it appears that we have a splice that we can use. (Note: to get the dotted trace, I set the plot Style for *y7* to Dot, and the plot resolution to 2).

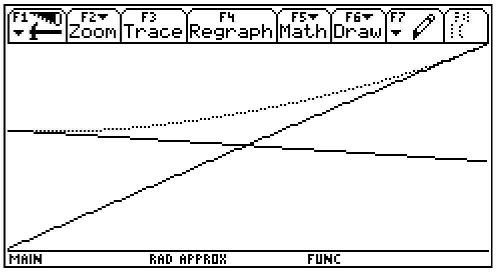
However, the splice is less than $f_1$ and $f_2$ over the entire range. Assuming that $f_1$ and $f_2$ accurately model the sin() function, we may prefer that the splice be above the two functions over the splice interval. This is accomplished by changing the value of y2. One reasonable value is the average of the values of $f_1(x_1)$, $f_1(x_2)$ and $f_2(x_3)$. This is easily calculated with

```
mean({y1(x2-h),y1(x2),y2(x2+h)})→yy2
```

We find and save the splice coefficients as before with

```
math\splice4(yy1,yy2,yy3,yd1,yd2)→spl4
```

and the splice now looks like this:



Note that the splice slopes still match the functions at the interval endpoints, but now the splice is above both functions.

### *Differentiating the splice*

To differentiate the splice, do not expand the splice polynomial and differentiate it: accuracy will suffer because of round-off error. Instead, first scale x to $x_s$, find $s'(x_s)$, then scale this result. In other words,

$$\frac{d}{dx}s(x) = \frac{1}{h}\cdot\frac{d}{dx_S}\left(a\cdot x_S^4 + b\cdot x_S^3 + c\cdot x_S^2 + d\cdot x_S + e\right) \qquad [20]$$

or

$$\frac{d}{dx}s(x) = \frac{1}{h}\cdot\left(4\cdot a\cdot x_S^3 + 3\cdot b\cdot x_S^2 + 2\cdot c\cdot x_S + d\right) \qquad [21]$$

where $\quad x_S = k\cdot(x - x2)$

The higher-order derivatives are

$$\frac{d^2}{dx^2}s(x) = \frac{1}{h^2}\cdot\left(12\cdot a\cdot x_S^2 + 6\cdot b\cdot x_S + 2\cdot c\right) \qquad [22]$$

$$\frac{d^3}{dx^3}s(x) = \frac{1}{h^3}\cdot(24\cdot a\cdot x_S + 6\cdot b) \qquad [23]$$

$$\frac{d^4}{dx^4}s(x) = \frac{24}{h^4}\cdot a \qquad [24]$$

The following function can be used to find the splice derivative of any order.

```
spli4de(x,x2,h,cl,o)
Func
©(x,x2,h,k,{list},order) 4th-order splice derivative
©9apr02/dburkett@infinet.com

© Input arguments
©
© x     The point at which to find the derivative
© x2    The splice interval midpoint
© h     The splice interval half-width
© cl    the list of splice function coefficients
© o     the order of the derivative; o > 0

local xs,a,b,c,d,k

cl[1]→a        © Extract polynomial coefficients
cl[2]→b
cl[3]→c
cl[4]→d
1/h→k          © Invert half-width to simplify later calculations
k*(x-x2)→xs    © Scale x

© Find derivative or return error string. The following when() is all on one line.

when(o≤0,"spli4de err",
when(o=1,k*polyeval({4*a,3*b,2*c,d},xs),
when(o=2,k^2*polyeval({12*a,6*b,2*c},xs),
when(o=3,k^3*(24*a*xs+6*b),when(o=4,k^4*24*a,0)))))

EndFunc
```

*spli4de()* returns the string "spli4de err" if the order *o* is less than 1. No testing is done to ensure the order is an integer. Note that derivatives with order greater than four are zero.

You should not, in general, use the splice function to estimate derivatives of order greater than the first, even though *spli4de()* provides that capability. Since we have not constrained the splice to match the higher order derivatives, they can vary wildly. In the example above, I fit a splice to two simple polynomials. The table below shows the derivative errors at the splice boundaries $x_1$ and $x_3$.

| Derivative order | f1'(x1) | Error at x1 | f2'(x3) | Error at x3 |
|---|---|---|---|---|
| 1 | 0.8529 | 6.09E-11 | 0.8558 | -3.6E-11 |
| 2 | -0.5316 | -6.06 | -0.6147 | -6.10 |
| 3 | -.9583 | 24,258 | 0 | -24,381 |
| 4 | 0 | -2.43E7 | 0 | -2.43E7 |

As expected, the first derivative errors at the splice boundaries are small, but the errors are so large for derivatives of orders two and greater as to make the derivative estimate useless.

### *Integrating the splice*

As with the derivative, we find definite integrals with the scaled polynomial and the scaled $x_s$ values. If

$$I = \int_{x_a}^{x_b} s(x)dx \qquad \text{and} \qquad I_s = \int_{x_{sa}}^{x_{sb}} s_s(x_s)dx_s$$

then $I = h \cdot I_s$

which is derived in a later section in this tip. We integrate the scaled splice function $s_s(x_s)$ with the scaled limits $x_{sa}$ and $x_{sb}$ found with

$$x_{sa} = \frac{1}{h} \cdot \left(x_a - x_2\right) \qquad x_{sb} = \frac{1}{h} \cdot \left(x_b - x_2\right)$$

The following function can be used to integrate the splice function.

```
spli4in(xa,xb,x2,h,cl)
Func
©(xa,xb,x2,h,{list}) 4th-order splice integral
©9apr02/dburkett@infinet.com

© Input arguments:
© xa   lower integration limit
© xb   upper integration limit
© x2   splice interval midpoint
© h    splice interval half-width
© cl   splice coefficient list


h*∫(polyeval(cl,xs),xs,(xa-x2)/h,(xb-x2)/h)

EndFunc
```

*spli4in()* does not check *xa* and *xb* to ensure that they are within splice interval. Results may be inaccurate when integrating beyond the interval limits.

### *Solving for the splice inverse*

Some applications require finding *x* such that s(x)=y, given *y*. This can only be done if the splice is strictly monotonic on the splice interval, that is, the splice s(x) has no minimum or maximum on the interval.

If the splice is monotonic, finding the inverse is a simple matter of using *nSolve()* as shown in the function below. We solve the scaled splice function, then un-scale the returned $x_s$ to find x. Since we are using the scaled splice function, we know that the solution $x_s$ is greater than -1 and less than 1, so we use those as the solution bounds.

```
spli4x(y,x2,h,cl)
Func
©(s(x),x2,k,{list}) 4th-order splice inverse
©9apr02/dburkett@infinet.com

© Input arguments:
© y    value of s(x) at which to find x
© x2   splice interval midpoint
© h    splice interval half-width
© cl   list of splice coefficients


(nSolve(polyEval(cl,xs)=y,xs)|xs≥⁻1 and xs≤1)*h+x2

EndFunc
```

This method is satisfactory if you only need a few values of the inverse. There is a faster method to find many values: find an inverse fourth-order splice to calculate *x* directly, given f(x). This is simple once we have found the original splice for y = f(x). We can use *splice4()* to find the inverse splice; we just exchange the x's for y's and correct the end-point derivatives:

```
splice4(xx1,xx2,xx3,xd1,xd2)
```

where *xx1* and *xx3* are the interval bounds, as before. For the scaling to work correctly, however, s(xx2) must be be halfway between *yy1* and *yy3*, so we find *yy2* with

$$yy1 = f_1(xx1) \qquad yy3 = f_2(xx3) \qquad yy2 = \frac{yy1+yy3}{2}$$

Given *yy2*, we solve for the corresponding *xx2* with *spli4x()* above:

```
spli4x(yy2,xx2x,hx,spl4x)→xx2
```

but note that *xx2x*, *hx*, and *spl4x* are the values for the *original* splice, that is, y = s(x). Find they y-axis half-width *hy* with

```
abs(yy2-yy1)→hy
```

then you can find the inverse derivatives with

```
hy*(𝑑(f1(x),x)|x=xx1)→xd1
hy*(𝑑(f2(x),x)|x=xx3)→xd2
```

We now have all the arguments for *splice4()*. To find x given s(x), use

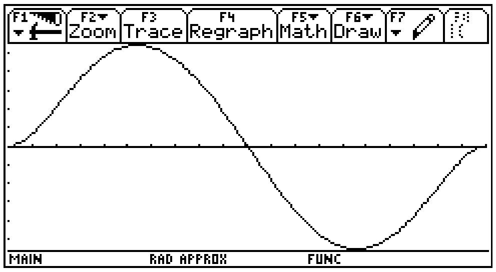
```
polyeval(spl4y,ky*(s(x)-yy2))
```

where ky = 1/hy.

This inverse polynomial is not the true inverse of the original splice. It is simply a polynomial which passes through the same three points as the splice, and has the same derivatives at the endpoints. If the splice is used for a narrow interval, the accuracy can be very good. Before using the inverse splice, you should check the accuracy. This is easily accomplished by graphing the error in the estimated x-values, found from

error = si(s(x)) - x

where s(x) is the splice function and si(x) is the inverse splice function. For the example shown above, this is the error in *x*:



The error in *x* is about ±2.4E-10, so this inverse is accurate to about 9 significant digits. Note that the error is zero at the endpoints and in the center, as it should be.

The function *spli4inv()*, shown below, finds the inverse splice by applying the steps described above.

```
spli4inv(f1,f2,x2,h,c)
Func
© ("f1(x)","f2(x)",x2,h,coeff_list)
© output: {a,b,c,d,e,y2,hy}
© 4th-order splice inverse polynomial
© 1ØmayØ2/dburkett@infinet.com
© calls math\spli4x(), math\splice4()

© Input arguments:
©
© f1    f1(x), passed as a string, for example "y1(x)"
© f2    f2(x), passed as a string, for example "y2(x)"
© x2    splice midpoint for s(x)
© h     splice half-width for s(x)
© c     list of splice coefficients for s(x)

© Output: {a,b,c,d,e,y2,hy}
©  where si(y) = a*ys^4 + b*ys^3 + c*ys^2 + d*ys + c
©  and ys =(y-y2)/hy
©  so find x = polyeval({a,b,c,d,e},(y-y2)/hy)


local yy1,yy2,yy3,yd,x1,x2,x3,xp1,xp3,hy
©
© x1    the original left interval bound; treated as si(y1)
© x2    s(x) such that x2=si(yy2)
© yy1   f1(x1)
© yy2   the mean of yy1 and yy3
© yy3   f2(x3)
© hy    y-interval half-width
©


expr("define f1(x)="&f1)      © Define local functions to be spliced
expr("define f2(x)="&f2)

x2-h→x1                       © Find splice interval bounds
x2+h→x3

f1(x1)→yy1                    © Find si(x1), si(x3)
f2(x3)→yy3
(yy1+yy3)/2→yy2               © Find splice interval midpoint
math\spli4x(yy2,x2,h,c)→x2    © Solve for x at interval midpoint
abs(yy2-yy1)→hy               © Find y-axis half-width
hy/(d(f1(x),x)|x=x1)→xp1      © Find dx/dy at x1 and x2
hy/(d(f2(x),x)|x=x3)→xp3

augment(math\splice4(x1,x2,x3,xp1,xp3),{yy2,hy})    © Solve for splice coefficients

EndFunc
```

Note that the output list includes *y2* and *hy* in addition to the coefficient list, since you will need them to evaluate the inverse polynomial. If you have saved the output list in *list1*, then use

```
left(list1,5)
```

to extract just the polynomial coefficients, and

```
list[6]
```

to get *y2*, and

```
list[7]
```

to get *hy*.

### *User interface program for splice4()*

The user interface program shown below, *spli4ui()*, automates the process of calculating a splice and checking the results. *spli4ui()* finds the splice coefficients and saves them. It also calculates and displays the errors for s(x) and the first derivatives.

*spli4ui()* uses a symbolic variable *ä*. If you have a variable *ä* in the current folder, it will be deleted.

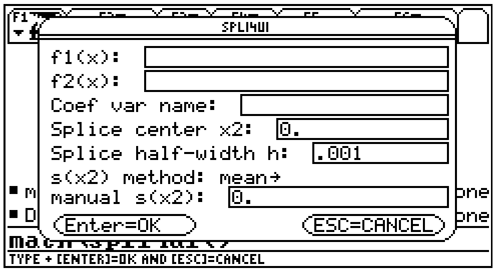
*spli4ui()* assumes that the derivatives of the functions to be spliced can be found with the built-in derivative function. For functions which cannot be differentiated by the calculator, you can still calculate a splice, but you cannot use *spli4ui()*.  See tip [6.26], *Accurate numerical derivatives with nDeriv() and Ridder's method*, to find the necessary derivatives.

```
spl4ui()
Prgm
©splice4() user interface
©16aprØ2/dburkett@infinet.com
```

```
© Calls math\splice4()

© Local variables
local
l,m,errx1,errx2,errx3,errd1,errd3,x1,äx2,x3,äh,yy1,äyy2,yy3,k,äf1,äf2,yp1,yp2,splc,äout

© l            drop-down selection variable
© m            drop-down selection variable

© äf1          f1(x)
© äf2          f2(x)

© x1           lower interval bound
© äx2          interval midpoint
© x3           upper interval bound

© yy1          f1(x1)
© äyy2         s(x2)
© yy3          f2(x3)

© äh           interval half-width
© yp1          f1'(x1)
© yp2          f2'(x3)
© k            scaling factor
© splc         splice coefficients list
© äout         name of coefficient list output variable

© errx1        splice error at x1
© errx2        splice error at x2
© errx3        splice error at x3
© errd1        derivative error at x1
© errd3        derivative error at x3

© Test for existing user input defaults variable spl4v; create if necessary
if gettype(spl4v)="NONE" then
 {"","","Ø.",".ØØ1","Ø.",""}→spl4v
endif

© Extract user input defaults
spl4v[1]→äf1
spl4v[2]→äf2
spl4v[3]→äx2
spl4v[4]→äh
spl4v[5]→äyy2
spl4v[6]→äout

© Prompt for user input
dialog
 title "SPLI4UI"
 request "f1(x)",äf1
 request "f2(x)",äf2
 request "Coef var name",äout
 request "Splice center x2",äx2
 request "Splice half-width h",äh
 dropdown "s(x2) method:",{"mean","center","manual"},m
 request "manual s(x2)",äyy2
enddlog
if ok=Ø:return                               © Return if [ESC] pressed

{äf1,äf2,äx2,äh,äyy2,äout}→spl4v             © Save user input defaults

expr(äx2)→äx2                                © Convert user input strings
expr(äh)→äh
expr(äyy2)→äyy2
expr("define äf1(x)="&äf1)                   © Create functions f1() and f2()
expr("define äf2(x)="&äf2)

1/äh→k                                       © Find scaling factor
äx2-äh→x1                                    © ... and splice interval bounds
äx2+äh→x3
```

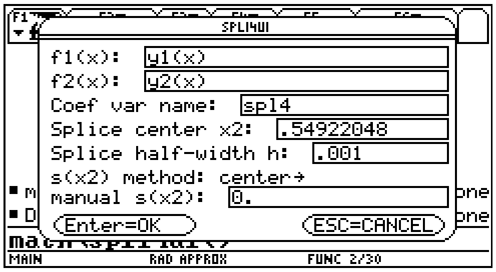
```
äf1(x1)→yy1                                  © Find splice bound y's
äf2(x3)→yy3

if m=1 then                                  © Set yy2 for s(x2) method:
(yy1+äf1(äx2)+yy3)/3→äyy2                    © ... 'mean' method
elseif m=2 then
 (äf1(äx2)+äf2(äx2))/2→äyy2                  © ... 'center' method
endif

d(äf1(ä),ä)|ä=x1→yp1                         © Find derivatives at interval bounds
d(äf2(ä),ä)|ä=x3→yp3

math\splice4(yy1,äyy2,yy3,äh*yp1,äh*yp3)→splc        © Find splice coefficients
splc→#(spl4v[6])                                     © Save coefficients to user variable

polyeval(splc,k*(x1-äx2))-yy1→errx1          © Find splice errors at bounds and midpoint
polyeval(splc,Ø)-äyy2→errx2
polyeval(splc,k*(x3-äx2))-yy3→errx3

k*(d(polyeval(splc,ä),ä)|ä=⁻1)-yp1→errd1     © Find derivative errors at bounds
k*(d(polyeval(splc,ä),ä)|ä=1)-yp3→errd3

clrio                                        © Display errors on Program I/O screen
disp "Absolute errors:"
disp "s(x1):  "&string(errx1)
disp "s(x2):  "&string(errx2)
disp "s(x3):  "&string(errx3)
disp "s'(x1): "&string(errd1)
disp "s'(x3): "&string(errd3)

EndPrgm
```

I will use *spli4ui()* to create the splice shown in the example above. On starting *spli4ui()*, the following dialog box is displayed:



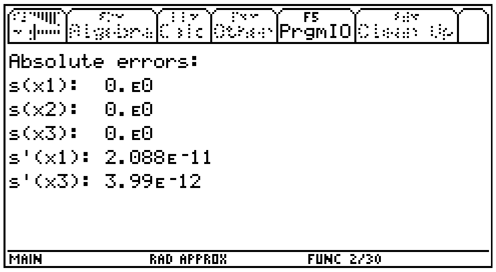
```
SPLI4UI
f1(x):
f2(x):
Coef var name:
Splice center x2:  0.
Splice half-width h:  .001
s(x2) method: mean→
manual s(x2):  0.
Enter=OK                ESC=CANCEL
TYPE + [ENTER]=OK AND [ESC]=CANCEL
```

This screen shot shows the dialog box with the parameters entered:



```
SPLI4UI
f1(x):  y1(x)
f2(x):  y2(x)
Coef var name:  spl4
Splice center x2:  .54922048
Splice half-width h:  .001
s(x2) method: center→
manual s(x2):  0.
Enter=OK                ESC=CANCEL
MAIN          RAD APPROX          FUNC 2/30
```

For f1(x) and f2(x), I can just enter y1(x) and y2(x), since I have previously defined these functions in the Y= Editor. I set the splice center and half-width, and chose 'center' for the *s(x2) method*. I need not enter a value for 'manual s(x2)', since I am not using the manual method. After pressing [ENTER], the errors are shown on the program I/O screen:



```
Absolute errors:
s(x1):  0.E0
s(x2):  0.E0
s(x3):  0.E0
s'(x1): 2.088E-11
s'(x3): 3.99E-12
```

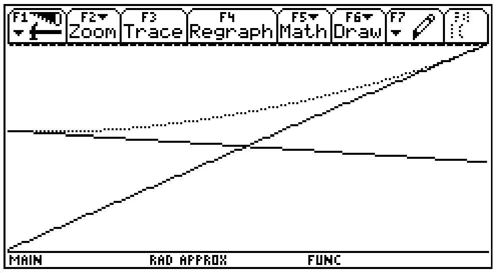
*spli4ui()* supports three options to set s(x2): center, mean and manual. With the 'manual' s(x2) method, you specify s(x2) in the 'manual s(x2)' field. The 'center' method sets

$$s(x_2) = \frac{f_1(x_1)+f_2(x_2)}{2}$$

so use the 'center' method to force the splice midway between two functions which do not intersect, or to force the splice through $f_1(x_2) = f_2(x_2)$. The 'mean' method sets

$$s(x_2) = \frac{f_1(x_1)+f_1(x_2)+f_2(x_3)}{3}$$

so use this method to force the splice through a point away from both functions. For example, this plot shows the splice and function differences with the mean method:



### *Scaling the derivatives*

Since the x- and y-data are scaled to avoid a singular matrix, we must also scale the derivatives of $f_1(x)$ and $f_2(x)$ to be consistent with the scaled *x* values. Starting with a general function

$$y = f(x)$$

and the general scaling equations $x_s = p \cdot x + q$ $y_s = r \cdot y + s$

we solve for *x* and *y*, $x = \frac{x_s - q}{p}$ $y = \frac{y_s - s}{r}$

substitute these expressions in the function definition $\frac{ys-s}{r} = f\left(\frac{xs-q}{p}\right)$

and solve for ys $ys = r \cdot f\left(\frac{xs-q}{p}\right) + s$

then take the derivative $\frac{dys}{dxs} = r \cdot \frac{d}{dxs} f\left(\frac{xs-q}{p}\right)$

or $\frac{dys}{dxs} = \frac{r}{p} \cdot \frac{d}{dxs} f(xs - q)$ [1A]

Now since $xs = p \cdot x + q$

$xs + dxs = p \cdot x + p \cdot dx + q = p \cdot x + q + p \cdot dx$

$xs + dxs = xs + p \cdot dx$

so $dxs = p \cdot dx$ [2A]

Replace [2A] in [1A] $\frac{dys}{dxs} = \frac{r}{p} \cdot \frac{d}{p \cdot dx} f(xs - q)$

or $\frac{dys}{dxs} = \frac{r}{p} \cdot \frac{d}{dx} f\left(\frac{xs-q}{p}\right)$

or simply $\frac{dys}{dxs} = \frac{r}{p} \cdot \frac{d}{dx} f(x)$

which is the relation we wanted. In this splice application, we scale only *x*, not *y*, so r = 1. In terms of the actual scaling variables $x_2$ and *h*, the scaling is

$$xs = \frac{1}{h} \cdot (x - x2) = \frac{1}{h} x - \frac{x2}{h}$$

so by inspection $p = \frac{1}{h}$

then $\frac{dys}{dxs} = h \cdot \frac{d}{dx} f(x)$

***Scaling the splice integral***

Given the scaled splice polynomial $s_s(x_s)$, we can easily find the definite integral

$$Is = \int_{xsa}^{xsb} ss(xs)dxs$$

but we really want the integral of the unscaled function:

$$I = \int_{xa}^{xb} s(x)dx$$

Using the definition of the definite integral: $Is = \lim_{n \to \infty} \sum_{m=1}^{\infty} ss(xs) \cdot \Delta xs_m$ [1B]

$$I = \lim_{n \to \infty} \sum_{m=1}^{\infty} s(x) \cdot \Delta x_m \qquad [2B]$$

where $$\Delta x_{sm} = \frac{x_{sb}-x_{sa}}{n} \qquad [3B]$$

$$\Delta x_m = \frac{x_b-x_a}{n} \qquad [4B]$$

Solve [3B] for *n*, replace in [4B], and define a new variable *c* such that

$$c = \frac{\Delta x_m}{\Delta x_{sm}} = \frac{x_b-x_a}{x_{sb}-x_{sa}} \qquad [5B]$$

then $$\Delta x_{sm} = \frac{\Delta x_m}{c} \qquad [6B]$$

Replace [6B] in [1B] for $$Is = \lim_{n \to \infty} \sum_{m=1}^{\infty} s_s(x_s) \cdot \frac{\Delta x_m}{c} \qquad [7B]$$

Now, since $s_s(xs) = s(x)$, for any given x and its equivalent scaled $x_s$, we move the 1/c constant outside the sum and limit, and [7B] becomes

$$I_s = \frac{1}{c} \lim_{n \to \infty} \sum_{m=1}^{\infty} s(x) \cdot \Delta x_m$$

which, on comparison with [2b], is just $Is = \frac{1}{c} \cdot I$

or $$I = c \cdot I_s \qquad [8B]$$

We can also express [8B] in terms of the original scaling factor *k*. The scaling is defined by

$$x_{sa} = \frac{1}{h}\left(x_a - x_2\right) \qquad x_{sb} = \frac{1}{h}\left(x_b - x_2\right)$$

which we can solve for *h*: $h = \frac{x_a-x_b}{x_{sa}-x_{sb}}$

and comparison with [5B] gives $h = c$

so we have the final desired result: $I = h \cdot I_s$

### [6.57] Sum binary '1' digits in an integer

Some applications need to find the number of '1' digits in a binary integer. For example, this sum is needed to calculate a parity bit. Another example would be a game or a simulation in which positions are stored as '1' digits in an integer, and you need to find the total number of pieces in all positions. The following TI Basic function will find the number of 1's in the input argument *n*.

```
sum1s(n)
Func
©(n) sum of binary 1's in n, n<2^32
©Must use Exact or Auto mode!
©26april02/dburkett@infinet.com
```

```
local t,k,s

{Ø,1,1,2,1,2,2,3,1,2,2,3,2,3,3,4}→t

Ø→s
for k,1,8
 s+t[(n and ØhF)+1]→s
 shift(n,⁻4)→n
endfor
return s

EndFunc
```

For example: `sum1s(Ø)` returns 0 `sum1s(ØhF)` returns 4
`sum1s(7)` returns 3 `sum1s(2^32-1)` returns 32

The last example results in a *Warning: Operation requires and returns 32-bit value*, but the correct result is returned. 32-bit integers are interpreted as 2's compliment signed integers, so the decimal range for valid input arguments is -2,147,483,648 to 2,147,483,647. This is 0h0 to 0hFFFF. (Recall that the TI-89/TI-92 Plus use *0b* to prefix binary integers, and *0h* to prefix base-16 integers).

*sum1s()* uses a table lookup (in the list *t*) to find the number of 1's in each 4-bit nibble of the input integer *n*. Each pass through the loop processes one nibble. The nibble is extracted by and-ing the current value of *n* with 0hF. I add 1 to the nibble value since list indices start at 1, not zero. After summing the correct list element, the input argument is shifted right four bits, so I can use the same bit mask of 0hF to extract the next nibble. The elements of list *t* are the number of 1s in all possible nibbles, in sequential order. For example, the nibble 0b0111 is decimal 7 which accesses the eighth element of t, which is 3.

There are many other ways to accomplish this task. A comprehensive survey of seven methods is the article *Quibbles and Bits* by Mike Morton (Computer Language magazine, December 1990). I chose the table lookup method because it has a simple TI Basic implementation. The number of loop iterations can be reduced by increasing the number of bits processed, but this increases the table size. We could process eight bits at a time in four loop iterations, but the table would have 256 entries. The method in *sum1s()* seems to be a good tradeoff between table size and loop iterations.

*sum1s()* executes in about 0.3 seconds/call. You could speed up *sum1s()* about 8% by making *t* a global variable and initializing it before running *sum1s()*. An even faster version would be coded in C. Unfortunately, a limitation in AMS 2.05 prevents this simple implementation, which would eliminate the *For* loop overhead:

```
sum(seq(t[(shift(n,-k*4) and ØhF)+1],k,Ø,7))
```

### [6.58] Extract floating-point mantissa and exponent

In some applications it is useful it obtain the mantissa and exponent of floating point numbers. In scientific notation, any floating-point number can be expressed as a.bEc, where *a.b* is the mantissa, *c* is the exponent, and *a* is not equal to zero. While the exponent and mantissa can be obtained starting out with the *log()* function, some book keeping is required to account for negative arguments. The function shown below returns the exponent and mantissa, and retains the full mantissa precision.

```
mantexp(n)
Func
©(n) return {mantissa,exponent}
©12mayØ2/dburkett@infinet.com

local s,e
```

```
format(n,"s")→s
expr(right(s,dim(s)-instring(s,"ᴇ")))→e
return {n*(1Ø^⁻e),e}

EndFunc
```

If the list returned by *mantexp()* is stored to foo, then use *foo[1]* to get the mantissa, and *foo[2]* to get the exponent.

Some examples:

| | | |
|---|---|---|
| `mantexp(Ø)` | returns | {0,0} |
| `mantexp(1)` | returns | {1,0} |
| `mantexp(1ØØ)` | returns | {1,2} |
| `mantexp(1.23E17)` | returns | {1.23,17} |
| `mantexp(-456E-1ØØ)` | returns | {-4.56,102} |

*mantexp()* works by converting the input argument to a string in scientific notation, then the exponent is extracted based on the location of the 'E' character in the string. The mantissa is found by dividing the original argument by 10^(exponent). While the mantissa could also be found from the string in scientific notation, two least-significant digits would be lost, since the *format()* function only returns, at most, 12 significant digits.

## [6.59] Accelerate series convergence

Most mathematical functions can be expressed as sums of infinite series, but often those series converge far too slowly to be of computational use. It turns out that there are a variety of methods that can be used to accelerate the convergence to the limit. Different methods are effective for different series, and an excellent overview is presented by Gourdon and Sebah in reference [1] below. As that paper is quite thorough, I will rely on it for background and just present a TI Basic program which implements one algorithm for accelerating convergence: Aitken's$\delta^2$ -process.

Aitken's method constructs a series of partial sums, which may converge faster then the original terms. It is basically an extrapolation process which assumes geometric convergence, so, the closer the convergence is to geometric, the better Aitken's method works. If $s_0$, $s_1$ and $s_2$ are three successive partial sums (not terms!) of a series, then Aitken's acceleration of the sum is

$$s' = s_0 - \frac{(s_0 - s_1)^2}{s_0 - 2 \cdot s_1 + s_2}$$

Some references give a different but algebraically equivalent form of the this equation, but this one is less susceptible to round-off errors, according to reference [3] below. You can actually repeat this process on the series of s' terms, but that usually doesn't increase the accuracy much.

```
aitkend2(l)
Func
©(terms list); returns {sn,delta}
©Accelerate series w/Aitken's δ^2 method
©11julØ2/dburkett@infinet.com

local i,n,s,sØ,s1,s2,s3,sa,sb

dim(l)→n

seq(sum(left(l,i)),i,1,n)→s          © Series of partial sums
```

```
s[n]→sØ                              © Save the individual sums we need
s[n-1]→s1
s[n-2]→s2
s[n-3]→s3

sØ-(sØ-s1)^2/(sØ-2*s1+s2)→sb         © Find the last sum
s1-(s1-s2)^2/(s1-2*s2+s3)→sa         © Find the next-to-last sum

return {sb,sb-sa}                    © Return the last sum, and the difference

EndFunc
```

*aitkend2()* is called with a list of series terms, and returns the last accelerated sum, and the difference between the last two sums. The difference is useful to get an idea of the convergence; if it is large compared to the sum, then convergence is poor. Try using more terms.

A much-used example for Aitken's method is this series, and I'll use it, too:

$$\frac{\pi}{4} = \sum_{k=0}^{\infty} \frac{(-1)^k}{2\cdot k+1} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \ldots$$

To create an input series of 11 terms for *aitkend2()*, use

```
seq((-1)^k/(2*k+1),k,Ø,1Ø)→s1
```

then

```
aitkend2(s1)
```

returns {0.7854599, 0.0001462}.

which results in an absolute error, compared to pi, of 0.000247. In comparison, the final difference of the original 11 terms is about 0.1, and the absolute error is also about 0.1. With the same 11 terms, Aitken's method improves the final result by nearly three orders of magnitude.

No convergence acceleration works well if the series terms are increasing. In this case, find the point at which the terms begin decreasing, sum the increasing terms, then apply the acceleration to the remaining terms.

References:

*[1] Convergence acceleration of series*, Xavier Gourdon and Pascal Sebah, 2002
http://numbers.computation.free.fr/Constants/Miscellaneous/seriesacceleration.html
A concise, thorough survey of convergence methods, starting with a good basic introduction. Covers seven different acceleration methods, including the (popular) Euler's method for series with alternating signs, and an interesting method based on the zeta function. Highly recommended.

*[2] Numerical Methods for Scientists and Engineers*, Richard W. Hamming, Dover, 1973.
A short but typically pithy description of Aitken's method, which Hamming oddly calls Shanks' method.

*[3] Numerical Recipes in Fortran*, 2nd edition, William H. Press et al, Cambridge University Press, 1992. http://www.nr.com/
Section 5.1 is aptly titled *Series and Their Convergence*, and Press and his cohorts first mention Aitken's method, but then provide code for Euler's method.

*(Credit to Ralf Fritzsch)*

### [6.60] Faster, more accurate exponential integrals

A post on the comp.sys.hp48 newsgroup announced that the HP-49G did not accurately find a numerical value for this integral

$$\int_{-3.724}^{\infty} \frac{e^{-x}}{x} dx$$

Except for a sign change, this is identical to the special function called the exponential integral:

$$\mathrm{Ei}(x) = -\int_{-x}^{\infty} \frac{e^{-x}}{x} dx$$

The TI-89 / TI-92 Plus also cannot find accurate values for this integral, and it is no wonder, since at the integrand singularity at x = 0, the limit is +infinity from the right and -infinity from the left. One expedient solution is a routine specifically designed to find the exponential integral:

```
ei(x)
Func
©(x) exponential integral Ei
©13julØ2/dburkett@infinet.com

local eps,euler,fpmin,maxit,k,fact,prev,sum1,term

6ᴇ⁻12→eps                             © Desired relative accuracy
.5772156649Ø153→euler                 © Euler's constant
1ØØ→maxit                             © Maximum iterations allowed for convergence
1ᴇ⁻9ØØ→fpmin                          © Number near floating-point minimum

if x≤Ø:return "ei arg error"          © Argument must be > Ø
if x<fpmin then                       © Handle very small arguments
 return euler+ln(x)
elseif x≤⁻ln(eps) then                © Use power series for x < 25.84Ø
 Ø→sum1
 1→fact
 for k,1,maxit
  fact*x/k→fact
  fact/k→term
  sum1+term→sum1
  if term<eps*sum1
   return sum1+ln(x)+euler
 endfor
 return "ei failed series"            © Return error string if convergence fails
else                                  © Use asymptotic expansion for large arguments
Ø→sum1
 1→term
 for k,1,maxit
  term→prev
  term*k/x→term
  if term<eps:goto l1
  if term<prev then
   sum1+term→sum1
  else
   sum1-prev→sum1
   goto l1
  endif
 endfor
 lbl l1
 return e^(x)*(1+sum1)/x
```

```
endif

EndFunc
```

I ported this algorithm, from *Numerical Recipes in Fortran*, to TI Basic. The input argument must be greater than zero, and error message strings may be returned.  You can check for this condition by using *getType()* on the result.

This table shows some results for the original example problem.

| Method | Result | Correct significant digits | Execution time |
|---|---|---|---|
| -ei(3.724) | -16.2252 9269 7647 | About 11 | 2 sec |
| `∫(ℯ^(-x))/x,x,-3.724,∞)` | -16.2967 8867 2016 | 2 | 134 sec |
| `nInt(ℯ^(-x))/x,x,-3.724,∞)` | -13.5159 0839 6945 | 1 | 53 sec |
| `nInt(ℯ^(-x))/x,x,-3.724,Ø) +`<br>`nInt(ℯ^(-x))/x,x,Ø,∞)` | -16.2967 8867 2016 | 2 | 56 sec |

It is interesting that the built-in `∫()` function is slightly more accurate that the purely numerical *nInt()*, perhaps because it symbolically finds the singularity at x = 0, and integrates over two ranges divided at x = 0.  Circumstantial evidence for this supposition is provided by that fourth method, which manually uses *nInt()* over the two ranges, and returns the same result as `∫()`.

Bhuvanesh Bhatt has also written a function for Ei(x), and you can get it at

*http://tiger.towson.edu/~bbhatt1/ti/*

This is a C function called *ExpIntEi()*, found in his C Special Functions package. You will need the usual hacks to use this as a true function on HW2 calculators; see the site for details.

For more information on the exponential integral, try

*Handbook of Mathematical Functions*, Milton Abramowitz and Irene A. Stegun, Dover, 1965. Section 5 describes and defines Ei(x), as well as its related integrals and interrelations. I used the table of numerical values to test *ei(x)*.

*Atlas for Computing Mathematical Functions*, William J. Thompson, Wiley-Interscience, 1997. Thompson calls Ei(x) the 'exponential integral of the second kind', and coverage begins in section 5.1.2. You can often get this book inexpensively from the bookseller Edward R. Hamilton, at *http://www.hamiltonbook.com/*

As mentioned, the algorithm for *ei(x)* comes from

*Numerical Recipes in Fortran*, 2e, William H. Press et al, Cambridge University Press, 1992. Section 6.3 covers the exponential integrals. This book is available on-line at *http://www.nr.com*.

You can also use *ei(x)* to find values for the logarithmic integral li(x):

$x > 1$

since li(x) = Ei(ln(x))

### [6.61] Find more accurate polynomial roots

*solve()* may return false solutions to ill-conditioned polynomials. An ill-conditioned polynomial is one in which small changes in the coefficients cause large changes in the roots. Since *zeros()* uses *solve()*, *zeros()* can return the same incorrect solutions. This tip gives an example, shows how to check the results and gives alternative methods for finding the better solutions. In general (and as usual), finding polynomial roots is not trivial, and the theoretic best attainable accuracy might be worse than you would expect, even for polynomials of relatively low degree.

For some examples in this tip, I define a 'best' solution for a root as *x* such that f(x+e) and f(x-e) are of opposite sign, or that one or both are zero, and *e* is the least significant digit of the floating-point mantissa. I will call this the *sign test*. There are better tests for root solutions, and I will discuss some of those as well.

The 'best' value for a polynomial root depends on what you want to do with it. Some common criteria for a root *x* are

1. *x* such that |f(x)| is as 'small' as possible. Ideally, |f(x)| would be zero, but this is unlikely with the finite resolution of floating-point arithmetic and its round-off errors.
2. *x* such that |f(x)| is as small as *required*. For problems based on physical measurements, it may be a waste of time to find the true minimum, since measurement error prevents such an accurate solution, anyway.
3. The polynomial coefficients can be reconstructed as accurately as possible from the roots. This is the same as saying that the errors in coefficients of the reconstructed polynomial are minimized. The polynomial is reconstructed with the roots $z_0$, $z_1$, ... $z_n$ as f(x) = $(x-z_0)(x-z_1)...(x-z_n)$
4. Some other various conditions for polynomial roots are met.

The conditions in criteria 4 may include these properties of polynomials:

$$\sum_{i=1}^{n} z_i = \frac{a_{n-1}}{a_n} \qquad \sum_{i>j} z_i z_j = \frac{a_{n-2}}{a_n} \qquad z_1 z_2 z_3 ... z_n = (-1)^n \frac{a_0}{a_n}$$

for this polynomial, with roots $z_1$, $z_2$, ... $z_n$

$$f(x) = a^n x_n + a^{n-1} x_{n-1} + ... + a_1 x + a_0$$

Or, we may want to use the roots to evaluate the polynomial in this form:

$$f(x) = a_n \left(x - z_1\right)\left(x - z_2\right)...(x - z_n)$$

The point is that different root-finding algorithms can return slightly different roots (or fail completely to find all or any roots), so, as usual, you really need to know why you want the roots to get useful results.

Getting back to the failure of *solve()*, I'll use this polynomial as an example:

$$y1(x) = x^3 + 4.217E17 \cdot x^2 - 3.981E20 \cdot x - 6.494E22 \qquad [1]$$

Define this polynomial in the Y= editor, then `zeros(y1(x),x)`

returns these candidate solutions: `{-4.217E17, -141.81969643465, 0}`

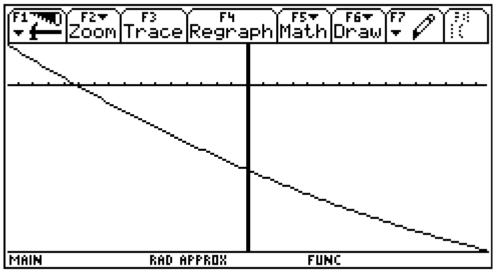
We find the function values at the solutions with `y1({-4.217E17, -141.81969643465, Ø})`

which returns (approximately) `{1.67879E38, 1E9, -6.494E22}`

These values may not seem 'close' to zero, still, they may be the best roots as defined above. The first step is to test each root. Command-line and TI Basic calculations on the TI-89 / TI-92 Plus are performed with a 14-digit mantissa, so we need to find the value of *e* for each root. If a root is expressed in scientific notation as a.bEc, where *a.b* is the mantissa, *a* is not equal to 0 and *c* is the exponent, then e = 10^(c-13). For example, for the first root of -4.217E17, e = 10^(17 - 13) = 1E4. The table below shows the verification results for all three roots.

| **root** | **y1(x-e)** | **y1(x+e)** |
|---|---|---|
| -4.217E17 | -1.610E39 | 1.946E39 |
| -141.8196 9643 465 | 7E9 | -3E9 |
| 0 | -6.494E22 | -6.494E22 |

Since the signs of y1(x-e) and y1(x+e) are different for the first two roots, they are at least plausible. However, the third root of zero fails the test. To verify that zero is not a root, we plot the function over the range of x = -200 to x = 200:



The graph shows the root at about x = -141.8, but x = 0 is clearly not a root.

The sign test is automated with the following function *proot_t1()*. The arguments are the list of polynomial coefficients and a list of at least one root, and it returns a list of boolean *true* and *false* results: *true* if the corresponding root passes the sign test, and *false* if it fails.

```
proot_t1(c,r)
Func
©(coefList,rootList)
©Sign test for polynomial roots
©Calls math\mantexp()
©14junØ2/dburkett@infinet.com

local i,o,e

{}→o                                 © Initialize result list
for i,1,dim(r)                       © Loop to test each root
 1Ø^(math\mantexp(r[i])[2]-13)→e     © Find least significant digit
 © Test passes if f(x)=Ø, or f(x-e), f(x+e) have opposite sign or are zero
 polyeval(c,r[i])=Ø or polyeval(c,r[i]-e)*polyeval(c,r[i]+e)≤Ø→o[i]
endfor

return o

EndFunc
```

This call tests the roots returned by *solve()*:

```
proot_t1({1,4.217E17,-3.981E2Ø,-6.494E22},{-4.217E17,-141.81969643465,Ø})
```

which returns {*true,true,false*}, indicating that the first two roots pass the sign test and the last root fails. Note the *proot_t1()* also tests for f(x) = 0 for each root, and returns *true* in that case.

*proot_t1()* can only be used to test real roots of polynomials with real coefficients. The method used to perturb the roots for the test does not make sense in the complex plane.

Another method to check polynomial solutions is to expand the roots into a polynomial and compare the resulting coefficients with the original coefficients. One way to do this is

```
expand(product(x-{zn, ... zØ}))
```

where {zn, ... z0} is a list of the roots and x is a symbolic variable. Applying this method to the roots above with

```
expand(product(x-{Ø,-4.217E17,-141.81969643465}))
```

gives

$$x^3+4.217E17\cdot x^2+5.9805365986492e19\cdot x$$

The two highest-order coefficients are the same, but the original constant term has disappeared and the coefficient for *x* has the wrong sign, let alone the right value. By now you should be starting to suspect that *solve()* has failed us.

You might be tempted to try *factor()* or *cFactor()* to find the roots, but they fail in the same way as *solve()*, returning (x+0) as one of the factors.

The following sections show a few methods to find the correct result for this root:

| | |
|---|---|
| Method 1: Invert polynomial coefficients | (routine *prooti()*) |
| Method 2: Laguerre's algorithm | (routine *polyroot()*) |
| Method 3: Eigenvalue methods | (routine *proots()*) |

These methods have trade-offs in speed, accuracy and code size, as well as their abilities to find all the real and complex roots.

*Method 1: Invert polynomial coefficients*

An interesting property of polynomials is that the roots map onto the reciprocals of the roots of the polynomial with the coefficients reversed, that is, if

$f_1(x)=a_3x^3+a_2x^2+a_1x+a_0$ and $f_2(x)=a_0x^3+a_1x^2+a_2x+a_3$

and the solutions to $f_2(x)$ are $z_1$, $z_2$ and $z_3$, then the solutions to $f_1(x)$ are $1/z_1$, $1/z_2$ and $1/z_3$. Sometimes solving this 'reversed' polynomial gives better solutions than the original polynomial. Using this method and *zeros()* with our example polynomial, we get

z1 = -141.8196 9643 465 (7E9, -3E9)

| | |
|---|---|
| z2 = -4.2169 9999 9999 9 E17 | (1.68E38, 3.72E39) |
| z3 = 1085.8557 4101 61 | (-5E10, 4.9E10) |

where the numbers in parentheses are y1(x-e) and y1(x+e). This method returns the correct first and third roots, but the second root fails the sign test.

If you save your polynomials in coefficient list form and evaluate them with *polyEval()*, it is easy to use the list reversal method in tip [3.4] to reverse the coefficients. Bhuvanesh Bhatt's MathTools package also include a list reversal function, *reverse()*.

The following function *prooti()* can be used to find the roots with the inversion method.

```
prooti(c)
Func
©(coefList) Poly roots by inversion
©11junØ2/dburkett@infinet.com

local ï

seq(c[ï],ï,dim(c),1,¯1)→c          © reverse coefficients
return 1/(zeros(polyeval(c,ä),ä))    © solve for roots and invert

EndFunc
```

*Method 2: Laguerre's algorithm*

Another alternative is to use a completely different algorithm to find the polynomial roots. *solve()* may recognize a polynomial and so use a specific algorithm to find its roots, in any event, it doesn't work for our example, and Laguerre's algorithm does. The function shown below, *polyroot()*, combines the code for the programs *laguer()* and *zroots()* from *Numerical Recipes in FORTRAN*. Call *polyroot()* with the polynomial coefficients as a list:

```
polyroot({an, ... aØ})
```

and the roots are returned as a list. You *must* set the Complex Format mode (with [MODE]) to RECTANGULAR before running *polyroot()*, since it uses complex arithmetic even for real roots. Also set the Exact/Approx mode to APPROX, or *polyroot()* will take an extremely long time.

*polyroot()* may return the string *"polyroot too many its"* if a solution does not converge in 80 iterations or less, for each root. The call for our example is

```
polyroot({1,4.217E17,-3.981E2Ø,-6.494E22})
```

which returns these roots in about 7 seconds:

```
{-4.217ØØØØØØØØØ1,1Ø85.855741Ø161,-141.81969643465}
```

All of these roots pass the sign test, and the coefficients are reconstructed to, at worst, 5 least significant digits.

Some comments on the operation of *polyroot()* follow this code listing.

```
polyroot(a)
Func
©({an,...,aØ}) return roots of polynomial
```

```
©Set Complex format to Rectangular
©12mayØ2/dburkett@infinet.com

local m,roots,eps,maxm,i,j,jj,ad,x,b,c,root


©-----------------------------------------------------------
© Define local function to find 1 root with Laguerre'e method

Define root(a,x)=Func
©({coeffs},guess) polynomial root
©NOTE: {a} is reversed!

local m,maxit,mr,mt,epss,iter,j,abx,abp,abm,err,frac,dx,x1,b,d,f,g,h,sq,gp,gm,g2

dim(a)-1→m                 © Find order of polynomial
2ᴇ⁻12→epss                 © Set estimated fractional error
8→mr                       © Set number of fractional values to break limit cycles
1Ø→mt                      © Set number of steps to break limit cycle
mt*mr→maxit                © Maximum number of iterations

{.5,.25,.75,.13,.38,.62,.88,1.}→frac      © Fractions used to break limit cycle

for iter,1,maxit           © Loop to find a root
 a[m+1]→b
 abs(b)→err
 Ø→d
 Ø→f
 abs(x)→abx
 for j,m,1,⁻1              © Evaluate the polynomial and first two derivatives
  x*f+d→f
  x*d+b→d
  x*b+a[j]→b
  abs(b)+abx*err→err
 endfor
 epss*err→err              © Estimate roundoff error in polynomial evaluation
 if abs(b)≤err then        © Return root if error condition met
  return x
 else                      © Execute Laguerre's method
  d/b→g
  g^2→g2
  g2-2.*f/b→h
  √((m-1)*(m*h-g2))→sq
  g+sq→gp
  g-sq→gm
  abs(gp)→abp
  abs(gm)→abm
  if abp < abm: gm→gp
  if max(abp,abm)>Ø then
   m/gp→dx
  else
   e^(ln(1+abx)+iter*i)→dx
  endif
 endif
 x-dx→x1
 if x=x1:return x          © Converged to root
 if mod(iter,mt)≠Ø then    © Every mt steps take a fractional step to break (rare) limit cycle
  x1→x
 else
  x-dx*frac[iter/mt]→x
 endif
endfor

return "polyroot too many its"    © May have this error for complex roots
EndFunc

©----------------------------------------------------------
© Begin polyroot()

dim(a)-1→m                        © Find order of polynomial
```

```
1ᴇ⁻12→eps                         © Set complex part limit
newlist(m)→roots                  © Create list to hold roots

seq(a[i],i,dim(a),1,⁻1)→a         © reverse coefficients for simpler looping

a→ad                              © Copy coefficients for deflation

for j,m,1,⁻1                      © Loop to find each root
 Ø→x                              © Set guess to zero to find smallest remaining root
 root(ad,x)→x                     © Find the root
 if gettype(x)="STR":return x     © Check for error from root()
 if abs(imag(x)) ≤ 2*eps^2*abs(real(x))   © Remove imaginary part of root if very small
  real(x)→x
 x→roots[j]                       © Save the root
 ad[j+1]→b                        © Perform forward deflation on remaining coefficients
 for jj,j,1,⁻1
  ad[jj]→c
  b→ad[jj]
  x*b+c→b
 endfor
 left(ad,j)→ad                    © Keep only deflated coefficients
endfor

for j,1,m                         © Polish roots with undeflated coefficients
 root(a,roots[j])→roots[j]
 if gettype(roots[j])="STR":return roots  © On error, return roots so far
endfor

return roots

EndFunc
```

*polyroot()* will also work for polynomials with complex coefficients.

I will not describe Laguerre's algoithm in detail, but you can find additional description in the references for Acton and Press below.

*polyroot()* first finds estimates for all the roots, then the roots are 'polished' to improve accuracy. For the estimates, *polyroot()* finds the smallest remaining root, then creates a new polynomial by dividing that root out of the polynomial. This process is called *deflation*. The new polynomial has degree of one less than the original, since one root is divided out. Finding the smallest roots first reduces errors from round-off; otherwise the small roots might disappear completely. Since deflating the polynomial can introduce rounding errors by changing the coefficients, the second pass of polishing the roots finds the roots again, this time using the original, undeflated coefficients, with the roots just found as initial guesses.

As with any feedback-based iterative method, it is possible for the algorithm to get stuck, oscillating between two solutions which do not meet the termination criteria. This oscillation is called a limit cycle, and *polyroot()* attempts to break limit cycles by periodically perturbing the current solution by some fractional amount. *mt*, *mr* and *frac* implement the cycle-breaking process such that a fractional step size from *frac* is taken every *mt* steps. As shown, the total number of iterations to find a root is 80. If a root isn't found in 80 iterations, *polyroot()* gives up and returns an error message. This would be unusual but may occur with complex roots. If this error occurs, *polyroot()* returns a list of the roots found up to the point of error, and the last element is the string "*polyroot too many its*". For example, if the program finds three roots 1, 2 and 3, but fails on the fourth root, the returned list is

```
    {1,2,3,"polyroot too many its"}
```

If you call *polyroot()* from another program, you can test for this condition with

```
    root[dim(root)]→r
```

```
getType(r)="STR"
```

*polyroot()* uses Laguerre's algorithm for both the initial root-finding and the polishing, however, this is not mandatory: you could use some other method for the polishing. The references at the end of this tip discuss this more and give some suggestions.

Since *polyroot()* uses complex arithmetic even to find the real roots, we have a dilemma I have not encountered before: when do we decide that a complex root is *actually* real, and only has a small complex part because of round-off error? It may be that the root really does have a small complex part, in that case we don't want to throw it away. Conversely, we don't really want to return a small complex component when the root is real. Press and his co-authors use this criteria, for a complex root of x = a + b*i:

$$\text{if } |b| \le 2 \cdot \text{eps}^2 \cdot |a| \text{ then } a \rightarrow x$$

*eps* is the desired relative accuracy of f(x). Press gives no justification for this criteria, and I could not derive it from several reasonable assumptions. So, I posted this as a question on the Usenet newsgroup sci.math.num-analysis. Mr. Peter Spellucci was kind enough to answer with this:

> *"I think this is one of the typical ad hoc "equal to zero within the roundoff level" decisions one often is forced to use. First of all, this makes sense only for a real polynomial, and I assume that this decision is applied simultaneously for the conjugate complex value. In Laguerre's method there appears the following:*
>
> *denom= p'+sign(p')*sqrt((n-1)*((n-1)*p'^2-n*p*p")) )*
>
> *and if the radicand is negative, it branches into the complex plane. Hence a sound decision would be based on the possible roundoff level in the evaluation of this expression, in the first position on the possible roundoff errors in the evaluation of p, p' and p". Bounds on this can be computed within the evaluation for example using Wilkinson's techniques. But this bounds may be too pessimistic and hence one usually relies on some rules of thumb. In the real case with a complex zero z, also conj(z) will be a zero. Multiplying out we get a quadratic factor*
>
> *x^2-2*re(z)*x+abs(z)^2*
>
> *and if this factor would not change within the computing precision by neglecting the imaginary part I would set it to zero. This however would mean |b| < |a|*sqrt(eps) , if eps represents the computing precision. Hence I wonder a bit about the settings you mention."*

There is quite a difference between Peter's criteria and that used in NRIF: compare 1.4E-6 to 2E-24, if we use *eps* = 2E-12. Peter's criteria, while theoretically quite sound, seems rather extreme in throwing away complex roots. At this point, I discovered that the authors of *Numerical Recipes* run a forum for this type of question, so I posted:

> *In the xroots() code on p367 of NRIF, a complex root x = a + bi is forced real if |b| <= 2*eps**2*|a|. I cannot derive this from a few different assumptions; the closest I can come is b^2 <= 2*eps*|a|, assuming that the root is forced real if |x| - |a| <= eps. How is this criteria derived?*

One of the authors of *Numerical Recipes*, Saul Teukolsky, answered:

> *The criterion in the Recipe, using eps**2, is purely empirical. After all, you may well have a root that has a very small imaginary part that is meaningful. But your criterion, with eps, is perfectly*

> *reasonable to use instead. It assumes that any small imaginary part cannot be determined with an accuracy better than eps relative to the real part.*

In the meantime, I had tested *polyroot()* with about 30 polynomials from textbooks and numerical methods books, and in all cases it returned both real and complex roots, as appropriate. While empirical, the criteria in *polyroot()* seems to work pretty well. Perhaps the best approach is to try the criteria now used in *polyroot()*, but if you can't find the roots you want, then try one of the other two criteria.

*Method 3: Eigenvalue method*s

We can find the roots of a polynomial by finding the eigenvalues of this *m* x *m* companion matrix:

$$\begin{bmatrix} -\frac{a_m}{a_{m+1}} & -\frac{a_{m-1}}{a_{m+1}} & \ldots & -\frac{a_2}{a_{m+1}} & -\frac{a_1}{a_{m+1}} \\ 1 & 0 & \ldots & 0 & 0 \\ 0 & 1 & \ldots & 0 & 0 \\ \ldots & \ldots & \ldots & \ldots & \ldots \\ 0 & 0 & \ldots & 1 & 0 \end{bmatrix}$$

where the eigenvalues will be the roots of the polynomial

$$P(x) = \sum_{i=1}^{m+1} a_i x^{i-1}$$

cybernesto has written a function called *proots()* which implements this method:

```
proots(aa)
Func
Local nn,am
dim(aa)→nn
aa[1]→am
right(aa,nn-1)→aa
list▸mat(aa/(⁻am))→aa
augment(aa;augment(identity(nn-2),newMat(nn-2,1)))→aa
eigVl(aa)
EndFunc
```

The input argument *aa* is the list of polynomial coefficients. For our example polynomial,

$$y1(x) = x^3 + 4.217E17 \cdot x^2 - 3.981E20 \cdot x - 6.494E22$$

*proots()* returns the candidate roots in about 1 second, compared to 6.4 seconds required for Laguerre's method as implemented in *polyroot()* above. Also, *polyroot()* is about 1773 bytes, while *proots()* is only 156. Unfortunately, the root at about -141.8 returned by *proots()* fails the sign test.

As another example, consider the polynomial

$$p(x) = (x+1)^{20} =$$

$$x20 + 20x^{19} + 190x^{18} + 1140x^{17} + 4845x^{16} + 15504x^{15} + 38760x^{14} + 77520x^{13} + 125970x^{12} +$$

$$167960x^{11} + 184756x^{10} + 167960x^{9} + 125970x^{8} + 77520x^{7} + 38760x^{6} +$$

$15504x^5 + 4845x^4 + 1140x^3 + 190x^2 + 20x + 1$

Obviously, there are 20 repeated roots of -1, but *proots()* returns (approximately) these roots in about 100 seconds:

-1.49339 ± 0.102038i
-1.40133 ± 0.280223i
-1.24788 ± 0.388669i
-1.08333 ± 0.41496i
-0.945259 ± 0.38717i
-0.848193 ± 0.295003i
-0.843498 ± 0.25989i
-0.750275 ± 0.212911i
-0.703918 ± 0.130283i
-0.682926 ± 0.043651i

If we substitute these roots in the original polynomial, we get 20 complex results near zero, but *proots()* has returned 10 complex root pairs, instead of the expected 20 real results. In about the same 100 seconds, *polyroot()* (Laguerre's method) returns the correct roots as a list of twenty 1's. The results from *proots()* are disturbed from the correct solutions by round-off error in calculating the eigenvalues of a 20 x 20 element matrix.

TI offers a flash application, *Polynomial Root Finder*, (I'll call it TI PRF) which also uses this eigenvalue method according to its PDF documentation. TI PRF also fails in a rather spectacular fashion with the 20-degree polynomial above, but with different results than *proots()*. The flash application returns two real roots of about -1.49821 and -0.869716, and 9 complex conjugate pairs. If these roots are evaluated in the polynomial, we again get real and complex results that are in the vicinity of zero. The magnitudes range from about 1E-6 to 2E-10.

The real drawback to these eigenvalue methods is that they give no estimate for the root errors, and no warning that the roots are wrong. Since we usually do not know in advance what the roots are, we might be led to believe that the returned roots were correct. The main advantage to *polyroot()* is that it attempts to find the roots to meet an error criteria, and, if it cannot, it returns an error message instead of a wrong solution.

*Comments*

This procedure can be used to find polynomial roots with a fair degree of success:

1. First, try *solve()* or *csolve()* (or *zeros()* or *czeros()*) to find the roots. If they work, they will be faster than *polyroot()*, and they can find some complex solutions which *polyroot()* cannot.
2. Check the solutions with the sign test method (*proot_t1()*), or by creating the original polynomial with the roots, or just by evaluating y(x) at each root. You can also plot the polynomial in the vicinity of the roots for additional assurance.
3. If the tests in step 2 fail, try *proots()* and repeat the tests.
4. If *proots()* fails, try *polyroots()*.

The limited floating-point resolution can result in an effect called destructive cancellation, the symptom of which is that the polynomial may not be strictly monotonic (increasing or decreasing) over a very narrow range of x. For this reason, the sign test is not always conclusive. Plotting f(x) - f(xc), where *xc* is a point in the center of the x-range, may help indicate this problem.

Solving polynomials is not trivial, particularly when the polynomial is ill-conditioned, and polynomials with multiple identical roots are always ill-conditioned. An ill-conditioned polynomial has a large dynamic range of coefficients, and the coefficient signs alternate. Forman S. Acton (reference below), has this to say about polynomial-solving systems in general:

> *"A system cannot do everything: There will always be some extremely nasty polynomials which will frustrate it and cause it to fail to find some of the roots. A system should handle most of the polynomials it is apt to encounter, but we cannot insist on perfection, lest the program become a monster that demands more than its share of library space and requires large amounts of running time looking for subtle difficulties that are not usually present. We must, however, insist that in the nasty situations the system should announce its difficulty, lest the trusting user be misled into thinking that all the roots have been found, and with the precision he desires."*

Considering that the TI-89 and TI-92 Plus are intended for educational applications, it is perhaps reasonable that it fail to solve our example polynomial, which is, however, taken from a problem in mechanical engineering (strength of materials). However, as Mr. Acton says, it is not acceptable that it returns a clearly incorrect root without a warning message.

This situation is somewhat puzzling considering that the TI-86 returns, very quickly, three correct solutions. The TI-89 / TI-92 Plus probably use a more general algorithm, while the TI-86 uses an algorithm specifically designed for polynomials.

Some references claim that it is better to normalize the polynomial coefficients by dividing them by the leading coefficient, so that coefficient becomes one. Other references claim that normalization introduces additional rounding errors, and so is best avoided.

Sometimes your polynomial may be in the form of an expression instead of a list of coefficients. There are available TI Basic which extract the coefficients from an expression with repeated differentiation. This is a clever technique which works great for symbolic coefficients, but it adds round-off error which will affect the solutions. The ideal solution would be to extract the coefficients with *part()*, which avoids any round-off error. I am working on such a program, but as it is not finished, the derivative method is better than nothing. Here is one version, written by J.M. Ferrard:

```
polyc(p,x)
Func
©(f(x),x) polynomial coefficients
©J.M. Ferrard

Local c,k
Ø→k

While when(p=Ø,false,true,true)
 p/k!|x=Ø→c[k+1]
 d(p,x)→p
 k+1→k
EndWhile

seq(c[k],k,dim(c),1,⁻1)

EndFunc
```

I have added the final *seq()* line so that the coefficients are returned in descending order of the independent variable powers. A call of

```
polyc(3*x^2+2*x-1,x)
```

returns {3,2,-1}

*References*

Polynomial root-finding is an important, well-developed topic in numerical analysis, so most books on numerical methods or numerical analysis include a section on finding polynomial roots. Those that have been most helpful to me follow.

*Numerical Recipes in FORTRAN; The Art of Scientific Computing*
Press, Teukolsky, Vetterling and Flannery, 1992, Cambridge University Press.
Section 9.5 is devoted to finding roots of polynomials, and describes the processes of deflation and inverting the coefficients. Some other methods are discussed, including the eigenvalue technique (used by cybernesto's *proots()*) and mention of the Jenkins-Traub and Lehmer-Schur algorithms.

*Numerical Methods that (usually) Work*
Forman S. Acton, 1990, The Mathematical Association of America.
Acton discusses polynomial root-finding in all of chapter 7, *Strategy versus Tactics*. He dismisses Laguerre's method as "not sufficiently compatible with our other algorithms', but discusses it, anyway. Mr. Acton's objection seems to be the requirement for complex arithmetic, but points out that Laguerre's method is guaranteed to converge to a real root when all coefficients are real.

*A survey of numerical mathematics, volume 1*
David M. Young, Robert Todd Gregory, 1988, Dover.
Chapter 5 is a thorough (70 page) development and examination of the entire process of finding polynomial roots and determining if the roots are 'good enough'. Starting with general properties of polynomials, developed in a theorem and proof format, Young and Gregory continue by examining the methods of Newton, Lin and Lin-Bairstow, and the secant method. They proceed to the methods of Muller and Cauchy before developing procedures for finding approximate values of roots, including Descarte's rule of signs, Sturm sequences, and the Lehmer-Schur method. Acceptance criteria for real and complex roots are derived. Matrix-related (eigenvalue) methods are described, including Bernoulli, modified Bernoulli and inverse-power (IP). The chapter continues with a discussion of polyalgorithms, which consist of two phases of root-finding: an initial phase, and an assessment/refinement phase. In the final section, other methods are very briefly discussed, including Jenkins-Traub and Laguerre, which the authors describe as "An excellent method for determining zeroes of a polynomial having only real zeroes ..."

*Rounding errors in Algebraic Processes*
J.H. Wilkinson, 1994, Dover Publications.
Wilkinson devotes chapter 2 to polynomials and polynomial roots, and develops several important ideas in solving polynomials in general. This book is a 'classic' in numerical analysis.

*Validating polynomial numerical computations with complementary automatic methods*
Phillipe Langlois, Nathalie Revol, June 2001, INRIA Research Report No. 4205, Institut National de Recherche en Informatique et en Automatique.
(This paper is available at http://www.inria.fr/index.en.html.)
While the focus of this paper is on using stochastic, deterministic and interval arithmetic methods to estimate the number of significant digits in calculated polynomial roots, section 3 summarizes the best attainable accuracy of multiple root solutions. This paper is also the source of the example polynomial $p(x) = 1.47x^3 + 1.19x^2 -1.83x + 0.45$.

The following web sites are useful or at least interesting:

*Mcnamee's bibliography on roots of polynomials*
http://www.elsevier.com/homepage/sac/cam/mcnamee/

No actual contents here, but an exhaustive bibliography which would be even more helpful with a little annotation. The categories are

> Bracketing methods (real roots only). Newton's method. Simultaneous root-finding methods. Graeffe's method. Integral methods, esp. Lehmer's. Bernoulli's and QD method. Interpolation methods such as secant, Muller's. Minimization methods. Jenkins-Traub method. Sturm sequences, greatest common divisors, resultants. Stability questions (Routh-Hurwitz criterion, etc.). Interval methods. Miscellaneous. Lin and Bairstow methods. Methods involving derivatives higher than first. Complexity, convergence and efficiency questions. Evaluation of polynomials and derivatives. A priori bounds. Low-order polynomials (special methods). Integer and rational arithmetic. Special cases such as Bessel polynomials. Vincent's method. Mechanical devices. Acceleration techniques. Existence questions. Error estimates, deflation, sensitivity, continuity. Roots of random polynomials. Relation between roots of a polynomial and those of its derivative. Nth roots.

*Polynomial sweep at the WWW Interactive Mathematics Server*
http://wims.unice.fr/wims/wims.cgi?session=RNC912CC8A.3&+lang=en&+module=tool%2Falgebra%2Fsweeppoly.en

This is a very illuminating animation of polynomial roots plotted in the complex plane. The polynomial is expressed parametrically, then the polynomial and its roots are plotted as the parameter varies. It is fascinating to watch the roots move as the polynomial changes. You can use example polynomials, or enter your own. Very cool!

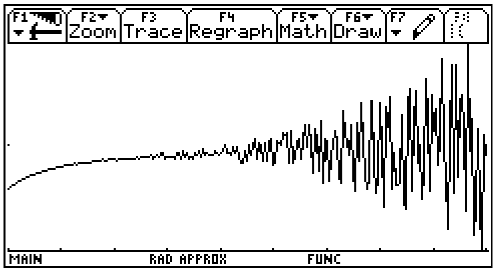
### [6.62] Try asymptotic expansion for functions of large arguments

This tip shows a function to generate an asymptotic series expansion for a function. As the name implies, the asymptotic series is useful for evaluating functions at large arguments. The present example shows that asymptotic expansion can fix a severe round-off error problem.

In June 2002, a post on the TI-89 / TI-92 Plus discussion group reported problems evaluating this function for large arguments:

$$f(x) = 1 - x + \frac{1}{2}\left(x - \frac{1}{x} + \frac{4}{6x+3(x-1)(x+1)\cdot\ln\left(\frac{2x}{x+1}-1\right)}\right) \qquad [1]$$

This function resulted from a tidal force analysis. Plotting the function from x = 100 to x = 1000 confirms there is definitely a problem:



The function starts getting erratic at about 370, and really gets chaotic at about x = 530. Since the limit of f(x) is one, there are definite problems in evaluating the function. As the problem occurs for large *x*, an asymptotic expansion is a potential solution. An asymptotic expansion is a series of the form

$$a_0+\frac{a_1}{x}+\frac{a_2}{x^2}+\frac{a_3}{x^3}+\ldots \qquad [2]$$

which is defined for sufficiently large *x*, if, for every *n* = 0, 1, 2, 3, ...

$$\left[f(x)-\left(a_0+\frac{a_1}{x}+\frac{a_2}{x^2}+\ldots\right)\right]x^n\to 0 \qquad \text{as} \qquad x\to\infty \qquad [3]$$

then

$$f(x)\approx a_0+\frac{a_1}{x}+\frac{a_2}{x^2}+\ldots \qquad [4]$$

Note that [2] may not converge for any *x*. In general, finding an asymptotic expansion is difficult, and many books have been written about the difficulties and some solutions. However, since we are only interested in a numeric approximation, our task is a little easier. Since [3] is defined for every value of n = 1, 2, 3, ..., we find $a_0$ by letting *n* = 0, then [3] becomes

$$f(x)-a_0\to 0 \qquad \text{or} \qquad a_0=\lim_{x\to\infty}f(x)$$

To find the second term $a_1$, we have

$$\left[f(x)-a_0-\frac{a_1}{x}\right]x\to 0 \qquad \text{or} \qquad a1=\lim_{x\to\infty}\left[f(x)-a_0\right]x$$

We can find as many terms as necessary by repeating this process, which is automated by this function *asympexp()*:

```
asympexp(fx,x,n)
Func
©(f(x),x,number of terms) Asymptotic expansion
©returns {an,...,aØ}
©4junØ2/dburkett@infinet.com

local i,j,a

{}→a

limit(fx,x,∞)→a[1]

for i,1,n-1
 limit(x^i*(fx-Σ(a[j+1]/x^j,j,Ø,i-1)),x,∞)→a[i+1]
endfor

seq(a[i],i,dim(a),1,¯1)

EndFunc
```

*asympexp()* should be run in Exact mode, since it used the *Limit()* function. *asympexp()* runs very slowly, but eventually returns these terms:

$$a_0=1 \qquad a_1=-\frac{3}{5} \qquad a_2=0 \qquad a_3=-\frac{4}{175}$$

$a_4 = 0$ $\quad a5 = -\frac{4}{375}$ $\quad a_6 = 0$ $\quad a_7 = -\frac{6364}{1010625}$

$a_8 = 0$ $\quad a_9 = -\frac{275708}{65690625}$ $\quad a_{10} = 0$ $\quad a_{11} = -\frac{20793692}{6897515625}$

$a_{12} = 0$ $\quad a_{13} = -\frac{-1335915116}{596288828125}$ $\quad a_{14} = 0$

$a_{15} = -\frac{7665058771652}{4288702777734375}$ $\quad a_{16} = 0$ $\quad a_{17} = -\frac{92763867329564}{64330541666015625}$

and approximate values for the non-zero terms are

$a_0$ = 1

$a_1$ = -0.6

$a_3$ = -2.2857 1428 5714 3 E-2

$a_5$ = -1.0666 6666 6666 7 E-2

$a_7$ = -6.2970 9338 2807 7 E-3

$a_9$ = -4.1970 6769 4210 6 E-3

$a_{11}$ = -3.0146 6399 3602 8 E-3

$a_{13}$ = -2.2785 9555 2080 3 E-3

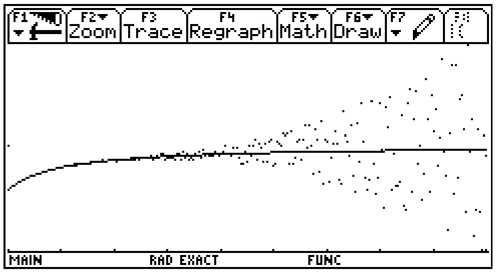
$a_{15}$ = -1.7872 6742 5349 8 E-3

$a_{17}$ = -1.4419 8797 2232 0 E-3

Note that *asympexp()* returns the coefficients as a list in the form {$a_n$, $a_{n-1}$, ..., $a_0$}, so the expansion can be evaluated as

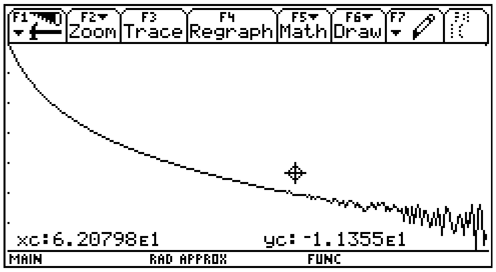
```
polyEval(a,1/x)
```

where *a* is the list of coefficients. In general, you would keep finding additional terms until they started to increase in value, then the best approximation would use all the still-decreasing terms. Even though the terms above are still decreasing, I decided to check the expansion with the terms through $a_9$. This plot shows the original function as points, the asymptotic expansion as a solid line, on the interval x = [100, 1000].



This is obviously a considerable improvement over the original function.

If we want to find both large and small values of the original function, we need to know the value of *x* at which to switch to the asymptotic function. One starting point is to plot the difference of the two functions, and switch to the asymptotic function before the round-off noise the original function begins. I could just plot the difference of the two functions, but the dynamic range is so large that the round-off noise would not be seen, so I plot the natural logarithm of the absolute value of the difference between the two functions, which essentially gives me a plot with a logarithmic y-axis. It looks like this:



xc:6.20798E1 yc:-1.1355E1

The plot shows the difference between the two function on the interval x = [5, 100]. The cursor readout shows that we could use x = 60 as the transition point between the functions. The function value at this point is about 0.989999885775, and the difference between the functions is about 6.3E-6. If we only need 5 or 6 significant digits in this range, the approximation is good enough. If we need more significant digits, we could increase the number of terms in the asymptotic expansion, but adding four more terms does not decrease the difference between the functions. We would need a lot more terms to increase the accuracy.

For more details on asymptotic expansions, you might start with section 18.6 of *Advanced Engineering Mathematics*, 6th edition, by Erwin Krezszig, Wiley, 1988. The entry for *asymptotic series* at Eric Weisstein's World of Mathematics site (*http://mathworld.wolfram.com/AsymptoticSeries.html*) also has a good bibliography.

### [6.63] Faster numerical matrix inverses

The calculator has a built-in matrix inverse operator, ^-1, but the C program *inv()*, by Rafael Humberto Padilla Velazquez, is much faster for inverses of matrices with floating-point elements. The call syntax is

```
inv(m)
```

where *m* is the matrix or matrix name. The matrix inverse is saved in the global variable *_asminfo*.

This table shows some typical execution times for random integer matrices of various sizes.

| Matrix size | m^-1 | Maximum residual | inv(m) | Maximum residual |
|---|---|---|---|---|
| 5 x 5 | 2.1 sec | 2E-13 | 1.8 sec | 1E-13 |
| 10 x 10 | 15.3 sec | 6.9E-13 | 3.1 sec | 1.4E-13 |
| 15 x 15 | 60.6 sec | 1.6E-12 | 11.4 sec | 7E-13 |
| 20 x 20 | 177.2 sec | 2.1E-12 | 28.6 sec | 6.8E-13 |

*inv()* is considerably faster than the built-in inversion, and the residuals slightly better, as well. The residual is the element with the largest absolute value of matrix *E*, where

$$E = M \cdot M^{-1} - I$$

*M* is the original matrix and *I* is the identity matrix. So, the residual is not actually the error in the inverse, but instead the 'round-trip' error in finding the inverse and evaluating the identity above.

### [6.64] Calculate through undef and get correct results

In some cases you can calculate with *undef* values and still get the correct result. Suppose we want to find the reciprocal of the sum of reciprocals of numeric list elements, like this:

```
1/sum(1/list)
```

where *list* is the list of elements. This calculation is used to find the equivalent resistance of a set of parallel resistors, where the list elements are the individual resistance values. Suppose *list* = {2,4,6,12}, then the expression returns 1 as expected. But if *list* = {2,4,6,12,0}, then the expression above returns the correct result of zero, even though manually performing the individual calculations results in *undef*:

```
1/list                                returns     {1/2, 1/4, 1/6, 1/12, undef}

sum({1/2, 1/4, 1/6, 1/12, undef})     returns     undef

1/undef                               returns     undef
```

Evidently, the calculator reorganizes the expression to avoid the division by zero.

# 7.0 TI Basic Programming Tips

### [7.1] Create evaluated Y= Editor equations in programs

You can create equations in the 89/92+ by storing the expression to the system variables yn(x), where 'n' is the number of the equation. For example,

```
a*x^2+b*x+c→y1(x)
```

stores the expression to the first equation, y1(x). But if *a*, *b* and *c* have numeric values, those values will not be substituted in the expression because the expression has not been evaluated. You can force the evaluation using *string()* and *expr(),* like this:

```
evalyx()
prgm
local a,b,c,ex

4→a
3→b
2→c

a*x^2+b*x+c→ex

expr("define y1(x)="&string(ex))

Endprgm
```

*evalyx()* puts this equation in the Y= Editor:

```
y1=4x²+3x+2
```

This method will work in a program, but not in a function. Since *a*, *b* and *c* were defined before the *expr()* function, those values are replaced. Any variables which are not defined will remain variables. Note that it is not necessary to define the *ex* variable; this works, too:

```
expr("define y1(x)="&string(a*x^2+b*x+c))
```

*(credit declined)*

### [7.2] Using the built-in function documentation in CATALOG

AMS 2.03 has a new feature in the CATALOG display in which user-defined programs and functions can be listed, in addition to the built-in functions. The functions are listed in alphabetical order, along with the folder they are in. This is very useful.

Even more useful is the fact that you can display a little documentation on the status line of the display. The status line is the very bottom line of the LCD display. This line will show the first comment in the program after the *Func* or *Prgm* statement. Also, if you press [F1], which is the Help menu tab, a dialog box opens that shows the same comment, but in a larger font which is much easier to read.

For example, these are the first few lines of a linear regression function:

```
linreghk(xl,yl,h,k)
func
©(xlist,ylist,h,k) return {b,a} for y=b*x+a through (h,k)
...
```

When this program is selected in CATALOG, the status line shows

```
©(xlist,ylist,h,k) return {b,a} for y=b*x+a through (h,k)
```

Consider using this feature to give the user all the critical documentation needed to use the function:

1. The input arguments for the function. In my example, the arguments are (xlist,ylist,h,k). I use parenthesis to indicate that these are the arguments. I use the names *xlist* and *ylist* to remind me that these are lists, and that they are the x- and y-data values for the regression. Note that the names used in the comment need not be the actual variable names; they can be more descriptive. Also, the phrase *through (h,k)* shows that *h* and *k* define a point.
2. The function output. In the example I use the phrase *returns {b,a}* to indicate that the functions returns two numbers in a list.
3. What the function actually does. The phrase *for y=b*x+a* shows that the function finds a linear regression function.
4. Any restrictions on the input variables. My example doesn't need this, but you could add a phrase like *h>0, k>0*.
5. The modes in which the function should be run.

It may be difficult to display all this information in the status line, but the dialog box will display 13 lines of about 30 characters each.

### [7.3] Using language localization

This section was written by Lars Fredericksen.

There are basically four problems connected to writing programs/functions which have to run under language localisation:

1. AMS versions previous to 2.0X do not support language localisation.
2. Programs and functions can only be compiled (tokenised) in the language they were written in.
3. The names of the variable types (getType(var)=”?”) are unknown at the time of programming, because they are language dependent.
4. The names of the modes are unknown (getMode(“?”)=”?”:SetMode(“?”,”?”) ). There are other mode related function where the same problem occur, but the handling is principal the same so I will concentrate on the mentioned functions.

1) In AMS version 2.0X the mode functions support a new syntax: a mode number can replace the mode name. It is a very important feature, because it provides a way of setting the mode without knowing the language dependent name. But it does also give problems with calculators running AMS previous to 2.0X, because they are not compatible with the new syntax. If programs have to run on all AMS version and under different languages it is necessary to take both models into account.

To test if the AMS supports language localisation can easily be done in a program with this code:

```
Try
 GetMode("1")
 True→V2XX
Else
 False→V2XX
 ClrErr
EndTry
```

The V2XX variable will contain True if language localisation is supported, or False otherwise. It is convenient if the V2XX variable is a global variable, so that it can be used by functions to test for the version.

2) Almost all function names are language dependent so programs/functions can only be tokenised in the language they are written. In other words, if you have written a program in English it can only be run if the language setting is set to English. But that is only the first time it is run. If the program is executed one time with the correct language setting, the language can be changed to any other language and the program will still work correctly. The program can even be edited in another language, just remember to run the program one time before changing language.  To save a language independent program, just make sure that the program is executed one time before transferring it with GraphLink.

3) Testing for a specific variable type in an unknown language using *getType()*

The following method can be used in programs/functions in all AMS versions.

```
Local TypeList,TypeMat
@ Acquiring the system for the type-name of a list.
{}→TypeList
getType(TypeList)→TypeList
@ Acquiring the system for the type-name of a matrix.
[Ø,Ø]→TypeMat
getType(TypeMat)→TypeMat
@ Testing if a variable is the type List.
If  getType(var)=TypeList Then
...
```

4) Setting/Testing a mode without knowing the language depending name.

When handling modes under language localisation only mode numbers should be used, because of some problem in the system with handling names containing foreign characters. This means is it necessary to know the mode numbering from Appendix D of the AMS 2.03 manual.

To set a mode and restore it:

```
Local Mode,OldMode
@ Set Graph=Function
If V2XX Then
 "1"→Mode
 SetMode(Mode,"1")→OldMode
ELSE
 "Graph"→Mode
 SetMode(Mode,"Function")→OldMode
Endif
...
SetMode(Mode,OldMode)
```

To test for a mode in a function is a little complicated. It is necessary for an installation program to get the language dependent mode names and store them in global variable, which can be used by the functions.

An example of translating Radian to the active language:

```
If V2XX Then
 "3"→MAngle
 SetMode(MAngle,"1")→OldMode
 GetMode(MAngle)→MRadian
 SetMode(MAngle,OldMode)
Else
 "Angle"→MAngle
 "Radian"→MRadian
Endif
```

When the modes have been translated and stored in global functions, they can be used in functions like this:

```
If GetMode(MAngle)=MRadian Then
...
```

*(Credit to Lars Fredericksen)*

## [7.4] Return error codes as strings

It is good programming practice, and it will save you and your user lots of grief, if you write your programs to return an error code if the program cannot run normally. As a simple example, consider a function that finds the reciprocal of the argument:

```
recip(x)
func
return 1/x
endfunc
```

If x=0, this function will return *undef*, which doesn't tell the user what went wrong. Also, if this function is called deep within a series of function calls, it can be difficult to determine just what caused the error.

This is a better method:

```
recip(x)
func
if x=Ø then
 return "bad arg in recip"
else
 return 1/x
endif
endfunc
```

Now the program checks for a valid argument before attempting the calculation. If the calculation can't be done, *recip()* returns a string instead of a number. The calling routine can use *GetType()* on the result to determine if an error occurred, and handle it appropriately. The returned string can be used as the error message displayed to the user, which tells him what went wrong (bad argument) and where it went wrong (recip).

This method fails if the function is designed to return a string instead of a number. In that case you may be able to use special coded strings for the error message. For example, suppose we have a routine that is supposed to build a string from two arguments:

```
stringit(a,b)
func
if dim(a)=Ø or dim(b)=Ø then
 return "©bad arg in stringit"
else
 return a&b
endif
endfunc
```

*stringit()* is supposed to return *a* concatenated with *b*, but only if both arguments have one or more characters. If this is not true, then *stringit()* returns an error message, the first character of which is the 89/92+ comment symbol, character 169. In this case, the calling routine checks the first character of the result to see if it is character 169. If so, the error is handled.

This method assumes that the strings will not start with character 169.

### [7.5] Considerations for referencing external programs

This is not so much as tip as it is some things to consider when you write a program that call routines which are external to that program. For example, suppose we have a program *A()* which calls program or function *B().* If both *A()* and *B()* are in the same folder, and that folder is the current folder, there is no problem. We have this:

```
A()                 Method 1
prgm
...
B()
...
endprgm
```

It is not so simple, however, when you consider some other likely cases. For example, B() might be a general-purpose utility program, located, for example, in the /utils folder. The most straightforward way to deal with this problem is to include the folder specification in the function call, like this:

```
A()                 Method 2
prgm
...
utils/B()
...
endprgm
```

This method has these disadvantages, because the folder name is hardcoded in program A():

- The *B()* utility must be in the */utils* folder, and it cannot be moved.
- The /utils folder cannot be renamed.
- An RAM or archive overhead is incurred each time *B()* is referenced in *A().*
- If you distribute your program to others, the */utils* folder must be created, and *B()* must be stored in it. Suppose that the user already uses that folder name, or already has a program of her own called *B()*? Then there is a conflict.

In many cases, the problem can be avoided by just keeping a copy of *B()* in the same folder as *A(),* and using method 1 above. This approach has these disadvantages:

- If *B()* is used my many programs, in many different folders, RAM or archive is wasted with each copy that is necessary.
- If changes are made to *B(),* you need to make sure that all of the copies are updated.

There is no clear solution to this problem. You need to weigh the advantages and disadvantages of each method. Most programmers use method 2, in spite of its disadvantages.

### [7.6] Recursion limits

The 89/92+ support recursion, whichmeans that a function can call itself. Since the operating system must save the state information of the calling program, there is a limit to the depth to which recursion can be used. I used this test routine to determine the maximum recursion calling depth:

```
rectest(k)
func
when(k=Ø,k,rectest(k-1))
endfunc
```

This program simply calls itself repeatedly, decrementing the *k* variable, until k = 0.

The program runs correctly for values of *k* of 41 or less. When *k* is 42 or greater, the program fails with a "Memory" error message. This limit does not seem to be sensitive to the amount of RAM and archive used. I did my testing with a TI-92 Plus, AMS version 2.05.

### [7.7] Use *return* instead of *stop* in programs for better flexibility, and to avoid a crash

*Note: this bug has been fixed in AMS 2.05. stop now works correctly in archived programs*

The *stop* instruction stops program execution. The *return* instruction, used without an argument, also effectively stops program execution. It is better to use *return* instead of *stop* for two reasons. First, it makes your programs more flexible. Second, under certain circumstances, *stop* can cause a crash that can be fixed only by resetting the calculator.

The improved flexibility comes about if you call one program from another program. For example, suppose you wrote a program called *app1()* which does something useful. Later, you decide it would be helpful to call *app1()* from another program. If you use *stop* in *app1(),* execution stops when *app1()* is finished, when you really want to return to the calling application.

Further, using *stop* instead of *return* can cause a calculator crash. The effect of this bug is to lock up the TI92 so that it does not handle keypresses and must be reset. I have only verified this bug on my TI92 with the Plus module installed, ROM version 1.05.

The bug occurs when an archived program containing the *stop* instruction is executed with the *expr()* instruction. To duplicate the bug, create this program in the \main folder:

```
stopit()
prgm
```

```
    stop
    endprgm
```

Archive *stopit(),* then, at the command line, enter

```
    expr("stopit()") [ENTER]
```

The BUSY annunciator in the LCD turns on and does not turn off. Keys seem to be recognized, but not handled. For example, pressing [green diamond] will toggle the 'diamond' annunciator, and pressing [2nd] will toggle the '2nd' annunciator. However, neither [green diamond][ON] nor [2nd][ON] will turn the calculator off. Pressing [ON] to break the program doesn't work, either. The calculator must be reset with [2nd][hand] + [ON].

This bug *does not* occur if *return* is used instead of *stop*.

The bug also occurs if *expr()* is used in a program, to execute the archived program containing the *stop* instruction. For example, this program

```
    stoptry()
    prgm
    expr("stopit()")
    endprgm
```

will also cause the bug, but only if *stopit()* is archived.

The bug will *not* occur if the program containing the stop instruction is not archived.

Note that the bug also occurs if *stopit()* is called indirectly using *expr(),* like this:

```
    app1()
    prgm
    expr("stoptry()")
    endprgm

    stoptry()
    prgm
    stopit()
    endprgm

    stopit()
    prgm
    stop
    endprgm
```

In this case, *app1()* is not archived, but *stoptry()* and *stopit()* are archived, and the bug occurs. And, in this case, *stopit()* is not called with *expr(),* but the routine that calls *stopit()* does use *expr().*

This bug is annoying because it can prevent you from implementing desired program operation. I have a complex application that uses an 'exit' program to clean up before the user quits the program. Clean-up includes deleting global variables, resetting the user's mode settings, and restoring the current folder at the time the application was called. This 'exit' routine is called from the application mainline, so I would like to use the *stop* instruction to terminate operation. There are several obvious solutions:

1. Leave the 'exit' routine unarchived. This consumes RAM for which I have better uses.

2. Call the 'exit' routine directly, without *expr().* This prevents me from using an application launcher I wrote to manage my apps.
3. Archive most of the exit program, but put the final *stop* instruction in its own, unarchived program which is called by the exit program. While this seems like an acceptable work-around, it should not be necessary.
4. Call the 'exit' routine as usual, but use *return* instead of *stop* to terminate program operation.

### [7.8] Return program results to home screen

A TI-89/TI-92 Plus *function* can return a result to the command line, but a *program* usually cannot. However, there are two methods to get around this limitation.

*Method 1: Samuel Stearly's copyto_h():*

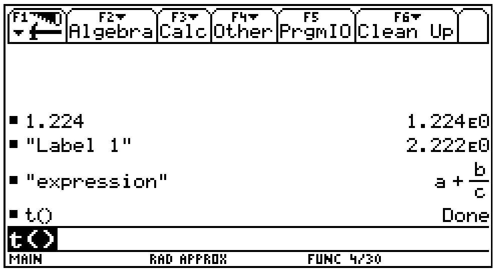
Samuel Stearly has written an ASM program that copies program results to the home screen. For the latest version, search for *copyto_h* on ticalc.org. The call syntax *for copyto_h()* is

```
copyto_h([name1[,name2]])
```

*name1* and *name2* are variable name strings. If no arguments are passed, *copyto_h()* displays the call syntax in the status line.  If *name1* is passed, the contents of *name1* are copied to the entry and answer areas of the home screen. If *name1* and *name2* are passed, the contents of name1 and name2 are copied to the entry and answer areas, respectively. This feature provides a very convenient way to label multiple results. *name1* and *name2* may include a folder specification. This simple test routine demonstrates the usage:

```
t()
Prgm
local v1,v2,v3,v4,v5

1.224→v1
"Label 1"→v2
2.222→v3
"expression"→v4
a+b/c→v5

util\copyto_h("v1")
util\copyto_h("v2","v3")
util\copyto_h("v4","v5")

EndPrgm
```

For this example, *copyto_h()* is installed in the *util* folder on the calculator, so the function calls specify that folder. After running *t()*, the home screen looks like this:



```
■ 1.224                      1.224E0
■ "Label 1"                  2.222E0
■ "expression"                 a + b/c
■ t()                           Done
t()
MAIN        RAD APPROX       FUNC 4/30
```

The first call to *copyto_h()* has only one argument (1.224), so that argument is placed in both the entry and answer columns of the history area. The next two calls have two arguments, so the first argument (the strings) is placed in the entry column, and the second argument is placed in the answer column. The last call shows that expressions are pretty-printed, as expected. Note that multiple calls to *copyto_h()* push each result up in the history display.

*copyto_h()* seems to run reliably on both HW1 and HW2 calculators, running AMS 2.05. I have had no problem with this routine, but there is always the risk that this type of program could cause unreliable calculator operation or a crash. I know of *no* problems that have been reported, and this program is so useful that I accept the small amount of risk that is involved. Mr. Stearly is to be congratulated for this accomplishment.

Mr. Stearly has also written:

*sendstr()* which works only with keyboard programs (*kbdprgmx()*) to send a string to the entry line:
http://www.ticalc.org/archives/files/fileinfo/166/16623.html

*copytobc()* which works with keyboard programs to copy an expression to the calculator clipboard:
http://www.ticalc.org/archives/files/fileinfo/146/14628.html

*Method 2: expr(result & ":stop")*

If the last line of your program is

```
expr(result & ":stop")
```

where *result* is a string, then *result* will be returned to the history area. For example,

```
expr("10"&":stop")
```

will return

```
10 : Stop
```

Now, this isn't exactly what we want, since the ":Stop" is tagged onto the result, but at least it works. The ":Stop" can be edited out.

Timité Hassan provides this routine *tohome()* to return results to the history area:

```
tohome(lres)
Prgm
Local zkk,execstr
""→execstr
If dim(lres)=0:return
For zkk,1,dim(lres)
 lres→wstr
 If instring(lres[zkk],"@")=1 then
execstr&string(mid(lres[zkk],2))&":"→execstr
 Else
  execstr&lres[zkk]&":"→execstr
 endif
Endfor
expr(execstr&":stop")
EndPrgm
```

*lres* is a list of strings to return to the history area. You can use the "@" character to add labels to the results. Note that the input string *lres* is stored to the global variable *wstr*, so that you can also recall *wstr* to get the results.

Some examples:

| | | |
|---|---|---|
| tohome({"45"}) | returns | 45 : : Stop |
| tohome({"45","x=6*po"}) | returns | 45 : x=6*po : : Stop |
| tohome({"@res1","45","@res2","x=6*po"}) | returns | "res1" : 45 : "res2" : x=6*po : : Stop |

cj points out that *tohome()* does not work if called from a keyboard program; those executed with [DIAMOND][n], where *n* is a number from 1 to 9. However, *tohome()* works if *kbdprgmn()* is called from the command line.

*(Credit to Timité Hassan and cj for Method 2)*

### [7.9] Passing optional parameters to functions and programs

Many programming languages support passing optional parameters to functions and programs, but TI BASIC does not. For example, you might have a function *f1()* that takes three or four parameters:

```
f1(p1,p2,p3,p4)
```

but if the user doesn't supply a value for the third parameter:

```
f1(p1,p2,,p4)
```

then the program supplies a default value for *p3*, or takes some other action. Note that this feature is supported in some built-in 89/92+ functions, such as *LinReg(),* for example.

You can simulate this operation to some extent in your own programs by passing the parameters as lists. For the example above, the call would be

```
f1(p1,p2,{p3},p4)
```

The *p3* parameter is always included in the call, but it may be an emply list. This can be determined if dim(p3) = 0. There are other variations on this theme, for example

```
f1(p1,{p2, p3, p4})
```

can be used to supply up to three optional parameters. Again *dim()* is used to determine how many parameters were supplied.

Alex Astashyn points out that there are some disadvantages to this method:

- You have to write additional code to recognize which arguments are supplied and to provide defaults.
- You have to remember to enter additional arguments in the list.
- You have to know what the default values are, to decide if you need to change them.

- You don't want to put the burden of remembering all this on the user, so it's easier to obligate the user to use the fixed number of arguments.

On the other hand, Frank Westlake notes these advantages:

- You can write one function instead of several very similar functions, which saves memory.
- The user doesn't have to remember which function to use for a given case.
- You can use an empty list as a request for help. If the list has no elements, the program returns a string that describes the list format.

*(Credit to Glenn Fisher, Alex Astashyn, and Frank Westlake)*

### [7.10] Calling built-in applications from your programs

It would be very useful to be able to call built-in 89/92+ applications from user programs. For example, if your program requires a matrix from the user, your program could call the matrix editor to create the matrix. This same idea applies to the text editor and the numeric solver.

In general, you can't do this. There is a work-around, but calling the built-in application must be the last action of the program, and you cannot return control to the program after the user is done with the application.

The basic idea is to use *setMode()* to set the Split 1 application, like this:

```
setMode("Split 1 App",appname)
```

where *appname* is the name of the application you want to run. Refer to the manual under *setMode()* for the applications that can be used.

For example, to run the numeric solver application, use

```
setMode("Split 1 App","Numeric Solver")
```

Note that you don't really have to be using the two split windows. This program example sets up an equation and calls the numeric solver:

```
t1()
Prgm
DelVar a,b,c
a+b+c→eqn
setMode("Split 1 App","Numeric Solver")
EndPrgm
```

*(Credit declined)*

### [7.11] Run TI Basic programs before archiving for better execution speed

Archiving TI Basic programs saves RAM, but you will get better execution speed if you run the program once before archiving it. As an example, I'll use TipDS' program to calculate the digits of pi:

```
PI(digits)
```

```
Func
©
©Programmed copyright by Tip DS
©This program is free for use, provided
©no modification is made.  There may be no
©charge for this program, unless
©permission is given in righting by
©Tip DS.  For more information, right
©to tipds@yahoo.com
©
Local  expan1,expan2,frac,answer,tmp1
x+Σ((⁻1)^t*x^(2*t+1)/(2*t+1),t,1,iPart(digits/(1.4))-1)→expan1
expan1|x=y→expan2
4*(4*expan1-expan2)|x=1/5 and y=1/239→frac
frac-3→frac
"3."→answer
iPart(10^digits*frac)→tmp1
answer&string(tmp1)→answer
EndFunc
```

Note that this is an *old* version of this program, and is only for demonstration purposes!

If you download this program with GraphLink, and immediately archive it, then it has a size of 520 bytes and an execution time of about 1.32 seconds per call. If you run the program once before archiving it, it has a size of 602 bytes, and an execution time of about 1.23 seconds per call, for an improvement of about 7%.

Bigger, more complex programs will show more improvement in execution speed. The reason is that if you archive the program before you run it, the operating system has to recompile the program each time it is run. If you run it once before archiving it, then the OS saves the compiled version.

On the other hand, if code size is more important to you than execution speed, you might want to archive the programs *before* running them. Note that the *PI()* program size increases by 82 bytes, or about 16%, if the program is run once before archiving.

If you are distributing a software package with many programs and functions, you might consider writing a routine that would automatically execute and archive all the programs for the user.

*(Credit to Lars Frederiksen)*

### [7.12] Access a variable from any folder with "_" (underscore)

Usually you refer to variables outside the current folder by specifying the folder name and the variable name, like this:

```
folderName\variableName
```

However, if you precede the variable name with an underscore, like this:

```
_variableName
```

you can use it from any folder without specifying its actual folder location.

The underscore is actually intended to be used to specify user-defined units. However, you can store numbers, strings, lists, expressions and matrices to this type of variable. Unfortunately, you cannot store functions or programs in these variables. The variable will appear in Var-Link in the folder in which it was created. You can create these variables as global variables in programs, but not functions. You cannot create them in functions.

You can execute a string stored in one of these variables like this:

```
"test2()"→_ttt1
expr(_ttt1)
```

This sequence will execute the program *test2().*

*(Credit to Frank Westlake)*

### [7.13] Write to program & function arguments

TI BASIC passes function and program arguments by value, not by reference. For example,

```
myprog(y)
```

actually passes the contents of the variable *y*, not the address of *y* or a pointer to *y. myprog()* can write to *y*, but if *y* is a global variable, *y* is not changed. For example, suppose you have a global variable *y* and you call this program, with *myprog(y):*

```
myprog(x)
Prgm
7→x
EndPrgm
```

The program runs without error, but the global variable *y* is not changed to 7, even though *x* contains 7 within the program.

To bypass this limitation, pass program argument variable names as strings:

```
myprog("y")
```

Use indirection within *myprog()* to access the variable:

```
myprog(x)
Prgm
local k
...
#x→k  ©Save y to k
...
k→#x  ©Save k to y
...
EndPrgm
```

This method works for programs, not functions, because functions cannot write to global variables. You can write to the function arguments as local variables, though, and indirection is not needed:

```
myfunc(x)
Func
...
x→k   ©Save y to k
...
```

```
k→x   ©Save k to y
...
EndFunc
```

The comments show what happens with the call *myfunc(y).*

### [7.14] Determine calculator model and ROM version in programs

It can be useful for your program to determine the model and ROM version of the calculator on which it is running. For example, since the 92+ LCD screen is 240 x 128 pixels, and the 89 screen is only 160 x 100 pixels, programs that take advantage of the larger 92+ screen won't work well on the 89: all the output won't be shown. The 92+ has many functions not included in the 92, so if your program uses those functions, it won't run on the 92.

If you want your programs to determine if they are running on an 89 or a 92+, Frank Westlake provides this code to identify the calculator model:

```
©Identify Model
Local j,tmp,timodel
list▸mat(getConfg(),2)→tmp
For j,1,rowDim(tmp)
 If tmp[j,1]="Screen Width" Then
  If tmp[j,2]=240 Then:"TI-92"→timodel
  ElseIf tmp[j,2]=160 Then:"TI-89"→timodel
  Else:""→timodel
  EndIf
 EndIf
EndFor
```

When this code finishes, the variable 'timodel' is "TI-92", "TI-89", or "" if the calculator seems to be neither model. Frank notes that this is slow, but reliable.

Lars Fredericksen offers this code which also determines if the calculator is a TI92 or TI92II:

```
©Identify model
Local j, tmp, timodel
 ""→timodel
 getconf()→tmp
 If getType(tmp)="EXPR" Then
   "TI-92"→timodel
 Else
  For j,1,dim(tmp),2
   If tmp[j]="Screen Width" Then
    If tmp[j+1]=240 Then
     "TI-92p"→timodel
    ElseIf tmp[j+1]=160 Then
     "TI-89"→timodel
    EndIf
    Exit
   EndIf
  EndFor
 endif
```

After execution, the local variable *timodel* holds the strings "TI-92" for a TI-92, "TI-92p" for a TI92 Plus, or "TI-89" for a TI-89.

Frank Westlake has found that the product ID can also be used to identify both the calculator model as well as the AMS ROM version. Specifically:

| Product ID | Calculator Model | AMS ROM version |
|---|---|---|
| 03-0-0-2A | TI-89 | 1.00 |
| 03-1-3-66 | TI-89 | 1.05 |
| 01-0-0-4A | TI-92+ | 1.00 |
| 01-0-1-4F | TI-92+ | 1.01 |
| 01-1-3-7D | TI-92+ | 1.05 |

Frank also provides these functions to identify the model and ROM version.

First, this is an example function which evaluates some user function *func1()* if the model is an 89 or 92+, and the version is 1.05 or greater. Otherwise, the function returns *undef*.

```
example()
func
local timodel
model()→timodel
if (timodel="TI-89" or timodel="TI-92+") and version()≥1.05:return func1()
return undef
endfunc
```

This function returns the product ID as a string, or "TI-92" if the calculator is a TI-92 without the Plus module:

```
pid()
func
© Product ID
local i,m
1→i
getconfg()→m
if gettype(m)="EXPR":return "TI-92"
while m[i]≠"Product ID"
i+1→i
endwhile
return m[i+1]
endfunc
```

This function calls *pid()* and returns the model number as a string.

```
model()
func
@Identify model
Local tmp
pid()→tmp
if mid(tmp,2,1)="1":return "TI-92+"
if mid(tmp,2,1)="3":return "TI-89"
return tmp
endfunc
```

This function calls *pid()* and returns the ROM version as a floating-point number.

```
version()
func
@Identify version
Local tmp
```

```
pid()→tmp
if mid(tmp,4,3)="Ø-Ø":return 1.Ø
if mid(tmp,4,3)="Ø-1":return 1.Ø1
if mid(tmp,4,3)="1-3":return 1.Ø5
return Ø
endfunc
```

Frank also offers these comments on his code:

*"In most cases none of this will be necessary. In the rare case that it is, it will probably be more useful to incorporate fragments into a single function or program making the overall code much smaller.*

*I claim no rights to any of this code, it is all public domain.*

*There doesn't appear to be any way to determine hardware version programatically."*

*(Credit to Frank Westlake and Lars Fredericksen)*

### [7.15] Avoid *For ... EndFor* loops

Loops are very slow on the 89/92+. The 89/92+ provide some functions which can be used to accomplish some operations that are traditionally done with loops:

- use *PolyEval()* to evaluate polynomials
- use *seq()* to generate a list of function values
- use *sum()* to add the elements of a list
- use sigma (Σ) to sum the values of an expression
- use upper-case pi (Π) to find the product of expression terms
- the common arithmetic operators (+, -, *, /, etc) operate on lists & matrices
- the 'dot operators' (.-, .*, ./, .^) operate on matrices and expressions
- the *submat()* function extracts part of a matrix

### [7.16] Use *when()* instead of *if...then...else...endif*

The 92+ manual describes using *when()* only to create discontinuous graphs, however, it is much more useful than that. It can be used in place of the *if...endif* construction, and is more compact.

*when()* functions can also be nested to create an *if...then...else..endif* structure. Suppose you have four functions f1(x), f2(x), f3(x) and f4(x). You want to evaluate the functions on intervals like this:

f1(x) when x < 1
f2(x) when x>= 1 and x < 2
f3(x) when x>= 2 and x < 3
f4(x) when x >= 3

The *If...EndIf* version looks like this:

```
if x < 1 then
 f1(x)
elseif x≥1 and x<2 then
 f2(x)
```

```
elseif x≥2 and x<3 then
 f3(x)
else
 f4(x)
endif
```

The nested-*when()* version looks like this:

```
when(x<1,f1(x),when(x<2,f2(x),when(x<3,f3(x),f4(x))))
```

The *if...endif* version is 107 bytes and executes in about 112 mS/call. The nested-when version is 73 bytes, and executes in about 100 mS/call. So, this method runs slightly faster and uses much less memory.

### [7.17] Returning more than one result from a function

Functions, by definition, can return only one result. You can get around this limitation by returning the answers in a list, string, matrix or data variable. The calling routine must extract the individual answers from the returned result. For example, to return the three results *a*, *b* and *c*, use

```
return {a,b,c}
```

then extract the individual results with

| | |
|---|---|
| `{a,b,c}[1]` | to get *a* |
| `{a,b,c}[2]` | to get *b* |
| `{a,b,c}[3]` | to get *c* |

Note that lists may contain not only numbers and expressions, but other lists and matrices if the elements are expressed as equality expressions, for example

```
return {x=[1,2],y={3,4},z={5,6,w={7,8}}}
```

The assignment variables *w*, *x*, *y* and *z* must not exist in the current folder, or the expressions will be evaluated and the list elements will becomre *true* or *false*. To retrieve the individual results, use the list indices and the *right()* function. For example, if the list above is stored in the variable *result*, then

| | |
|---|---|
| `right(result[1])` | returns [1,2] |
| `right(result[2])` | returns {3,4} |
| `right(result[3])` | returns {5,6,w={7,8}} |
| `right(right(result[3])[3])` | returns {7,8} |

To minimize the chance that the assignment variables exist, you can use international characters. The assignment variables need not be unique, for example, this works:

```
return {ä=[1,2],ä={3,4},ä={5,6,ä={7,8}}}
```

Another method to return more than one result is to use the results as arguments of an *undefined* user function. For example, if your function uses

```
return udf(10,20,{1,2,3})
```

then

```
udf(10,20,{1,2,3})
```

will be returned, as long as *udf()* is not actually defined as a function. You can use the *part()* function to extract the various results. With the expression above,

| | |
|---|---|
| `part(udf(10,20,{1,2,3}),1)` | returns 10 |
| `part(udf(10,20,{1,2,3}),2)` | returns 20 |
| `part(udf(10,20,{1,2,3}),3)` | returns {1,2,3} |

In general, for *part(exp,n)*, the *n*th argument is returned.

*(Credit to Glenn E. Fisher and Bhuvanesh Bhatt)*

### [7.18] Simplest (?) application launcher

Here's the problem: as I use my 92+ more and more, I write and download more programs. Considering that I keep each application in its own folder, and the 8-character name limit prevents really memorable names, I can't remember where a program was, or what it was called. The problem is solved with an application launcher, which shows intelligible descriptions of the programs, then executes the one I choose.

This tip shows two application launchers. I wrote the first one, which sets the application folder. Daniel Lloyd wrote the second one, which is smaller. Daniel's method does not set the application's folder, which doesn't matter if the application doesn't need it, or if the application sets the folder itself.

*The first application launcher*

The code below shows a simple application launcher.

```
apps()
Prgm
local k,appdesc,appfold
setfold(main)
mat▸list(submat(appsdef,1,1,rowdim(appsdef),1)→appdesc
popup appdesc,k
appsdef[k,2]→appfold
setfold(#appfold)
expr(appfold&"\"&main\appsdef[k,3])
setfold(main)
EndPrgm
```

*apps()* displays a pop-up box from which I select the program by its description. *apps()* then sets the current folder to the application folder, executes the program, then sets the current folder to *\main* when the program is done.

The application information is stored as strings in a 3-column matrix named *appsdef*. Use the matrix editor to create and edit this matrix. The three columns are:

column 1: Application description
column 2: Application folder
column 3: Application program name

A typical *appsdef* matrix might look like this:

| c1 | c2 | c3 |
|---|---|---|
| "Voltage divider solver" | "voltdiv" | "vdiv()" |
| "Cubic spline" | "cubspline" | "spline()" |
| "RTD resistance" | "rtdeqs" | "RTD385()" |

For example, the program *vdiv()* is located in folder *voltdiv*. The description that will be shown in the pop-up box is "Voltage divider solver".

*The second application launcher*

Daniel's application launcher looks like this:

```
kbdprgm1()
Prgm
setFold(main)
Local a
PopUp mat>list(pmat[1]),a
expr(pmat[2,a])
EndPrgm
```

Since the program is named *kbdprgm1*, it can be launched from any folder by pressing [DIAMOND] [1]. You could also rename my *apps()* program to *kbdprgm1*, with the same effect.

In this program, the application information is saved in a 2-row matrix called *pmat*. One column is used for each application. For the example applications above, *pmat* looks like this:

| c1 | c2 | c3 |
|---|---|---|
| "Voltage divider solver" | "Cubic Spline" | "RTD resistance" |
| "voltdiv\vdiv()" | "cubspline\spline()" | "rtdeqs\RTD385()" |

The first row shows the text that will appear in the pop-up menu. The second row specifies the folder and the application name, including the parentheses.

Daniel's launcher requires a simpler matrix to save the application information. Also, since he saves the pop-up menu items as the *row* of the matrix, instead of the column, he can simply use

```
PopUp mat>list(pmat[1]),a
```

to create the pop-up menu, since pmat[1] accesses the entire first row of the matrix. Daniel's launcher is also smaller because he doesn't set the folder to *\main* after the application executes. This isn't really necessary, but you can add it, if you want it.

*(Credit for second method to Daniel Lloyd)*

### [7.19] Bypass programs locked with ans(1) and 4→errornum:passerr

It is possible to prevent viewing TI BASIC source code. First, put this near the beginning of the program:

```
ans(1)
```

From the command line, execute:

```
4→errornum:passerr
```

then run the program. Now, you cannot view the program beyond the ans(1). Further, if you try to edit the program, all the 'invisible' code is lost. If you want to look at the source of a program 'locked' like this, to determine if it may contain a virus or poor programming practices, Frank Westlake provides this work-around:

1. Make a copy of the TI program on your computer so you won't have to remember how to undo the changes (it may not run with the changes).
2. Open the copy with a PC hex editor and go to the 14th byte from the end of the file. Count the last byte as one, second to the last as two, etc., until you get to 14.
3. Change byte 14 to hexidecimal 87, and change byte 13 (next one towards the end) to hexidecimal 3A.
4. Save the file and open it in Graphlink.
5. If there doesn't appear to be anything to worry about, delete the copy and load the original.

Frank also says:

*"I just used this procedure to check a whole groupfile of programs and if worked well. When I edit disabled one of my own programs I had to use a more complicated method, so you might have to experiment some. A knowledge of the file structure is extremely helpful so you may want to download a document describing it from one of the archive sites. It also helps to compare the file to one that isn't disabled. If I get time later this summer I'll try to come up with something more reliable."*

*(Credit to Frank Westlake)*

### [7.20] Running programs within a program

Suppose you have a program name stored as a string in variable *pname*. You can execute that program like this:

```
expr(pname)
```

Note that the *pname* string must include the parenthesis. For example, if the program name is *foo*, then *pname* is "foo()"

If the program is not in the current folder, you need to specify the folder like this:

```
expr(fname&"\"&pname)
```

where *fname* is the folder name string. If the folder name is known at execution time, you can just use

```
expr("foofold\"&pname)
```

where "foofold" is the folder where the program in *pname* resides.

This is useful when you want to execute one or more programs from your program, but you don't know in advance which program.

### [7.21] Use *undef* as an argument

*undef*, which means 'undefined', can be used as an argument in some functions and commands. For example, if this expression is entered in the Y= editor,

```
y1(x)=when(x<2,when(x<Ø,undef,x^2),undef)
```

then x^2 is plotted only for x>0 and x<2.

*(Credit to Rick A. and Ray Kremer, submitted by Larry Fasnacht)*

### [7.22] Local documentation for functions and programs

As you collect more programs on your calculator, it can be difficult to remember what the program does, or what arguments it takes, or what limitations it has. Frank Westlake shows a method to embed the documentation as comments within the program itself, such that it can be displayed with the 'Contents...' command in Var-Link. Check here for more details:

*http://members.web-o.net/westlake/ti/intdoc.html*

The basic idea is to use char(10) and char(12) to format the text to make it easy to read.

### [7.23] Passing user function names as program/function arguments

You cannot directly pass user functions as arguments to programs or functions, because the 89/92+ will try to evaluate them when the function is called. Many routines require a function as an input.

The general method is to pass the function name as a string. The called function then evaluates the function using either *expr()* or indirection.

*Using expr()*

To pass a function successfully, pass the name and argument variables (or variables) as a string, like this:

```
t("f1(x)","x")
```

where *f1(x)* is the function to be evaluated, and *x* is the function argument variable. To evaluate this function in program *t(),* build a string and evaluate the string with *expr().* We want a string of the form

f1(x)|x=xval

where *xval* is the value at with to evaluate the function. So, the complete program looks like this:

```
t(fxx,xx)
Prgm
local xval,result
⁻1Ø→xval
expr(fxx&"|"&xx&"="&string(xval))→result
EndPrgm
```

Usually the function must be evaluated several times, and there is no need to build the entire string expression each time. Avoid this by saving most of the expression as a local variable:

```
t(fxx,xx)
Prgm
local xval,result,fstring
fxx&"|"&xx&"="→fstring
1Ø→xval
expr(fstring&string(xval))→result
EndPrgm
```

Most of the string is saved in local variable *fstring*, which is then used in *expr().*

*t()* can be be a function or a program.

### *Using indirection*

Instead of using *expr(),* you can use indirection. The function name must still be passed as a string, but the parameters can be passed as simple expressions.  For example, suppose we have a function *t3()* that we want to evaluate from function *t1(). t3()* looks like this:

```
t3(xx1,xx2)
Func

if xx1<Ø then
 return ⁻1
else
 return xx1*xx2
endif

EndFunc
```

This is the calling routine routine *t1():*

```
t1(fname,x1,x2)
Func

#fname(x1,x2)

EndFunc
```

Then, to run *t1()* with function *t3(),* use this call:

```
t1("t3",2,3)
```

which returns 6.

*When passing just the function name DOES work:*

Sometimes you *can* pass the function and variable, and it will work. For example, suppose we use the call

```
t(f1(x),x)
```

with this program

```
t(fxx,xx)
Prgm
local xval,result
1Ø→xval
fxx|xx=xval→result
EndPrgm
```

In this case, the calculator will try to evaluate *f1(x)* when *t()* is called. If *x* is undefined, *f1()* will be symbolically evaluated (if possible) with the variable *x*. This expression is then passed as *fxx*. Later in the line fxx|xx=..., the symbolic expression is evaluated. For example, suppose that *f1(x)* is defined as

```
f1(x)
Func
2*x-3
EndFunc
```

so, the first evaluation (when *t()* is called) results in fxx = 2*x-3, which is then correctly evaluated later. However, if the function performs a conditional test on *x*, the program will fail. For example, if we have

```
f1(x)
Func
if x>Ø then
 2*x-3
else
 2*x+3
endif
EndFunc
```

the value of *x* is not defined at the time of the *t(fxx,xx)* call, so the program fails with the error message *A test did not resolve to TRUE or FALSE*.

*(credit for indirection method to Eric Kobrin)*

### [7.24] Use a script for self-deleting set-up programs

In general, a TI Basic program or function cannot delete itself. However, scripts (text files) can delete themselves, and this can be used to create a self-deleting set-up process. For example, suppose that the set-up program is called *setup(),* and the set-up script is called *setscrpt*. *setscrpt* looks like this:

```
:Press [f2] [5] to run setup
C:setup()
C:Delvar setup()
C:Delvar setscrpt
```

The first line describes how to run the setup script. The second line runs the *setup()* program. The third line deletes the set-up program, and the fourth line deletes the set-up script.

The set-up script can be created in the text editor or in GraphLink.

Christopher Messick adds these comments:

> *I noticed through accident that TI-89 text files have a self-destructive property (they can delete themselves). The above script runs the setup program, then deletes the program, then will delete itself from memory. It can do this because it doesn't execute directly from the text editor itself, but temporarily switches to the home screen "application," and pastes the command to the command line to execute the it. Afterwords, it returns to the text editor.*

*(credit to Christopher Messick)*

### [7.25] Use scripts to test program and functions

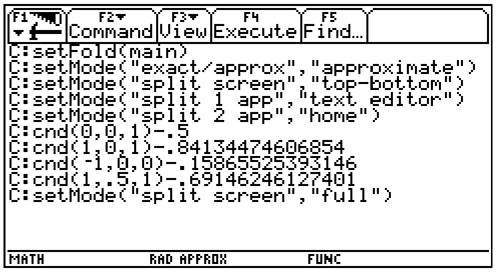
Scripts are text files that include commands that you can execute from the text editor. See the *TI89/92+ Guidebook*, page 328 for details. Also see page 94 for instructions to save the home screen as a text file, to be used as a script.

This can be very useful as you are testing and debugging functions with different arguments. You can enter each function call as a command in the script file. This also makes 'regression testing' easier. Regression testing is a software engineering term that refers to verifying that a function still works properly after changes have been made. Suppose that you have written a function and tested it, and you think that it is correct. However, after using it for a while, you find that it does not work for some arguments. If you have saved the test calls as a script file, you can easily verify that your corrections do not affect the parts of the program that *did* work, before.

You can also use commands in the script file to change the modes (Auto, Exact, Radians, Complex, etc.) and change folders, if necessary. Any command that can be executed from the command line can be included in the script file.

Using script files for testing works best if the Split screen display mode is used. This way, you can see both the script file commands and the results.

As an example, suppose I have written a function to find the cumulative normal distribution called *cnd()* in my *\main* folder. This function takes three arguments and returns a single numeric result. I want to test it with a variety of input argument combinations, and compare the results to those I have found from another source. I use the text editor ([APPS] [8]) to create a new text file with the name *cndtest*. The text file looks like this:



```
C:setFold(main)
C:setMode("exact/approx","approximate")
C:setMode("split screen","top-bottom")
C:setMode("split 1 app","text editor")
C:setMode("split 2 app","home")
C:cnd(0,0,1)-.5
C:cnd(1,0,1)-.84134474606854
C:cnd(-1,0,0)-.15865525393146
C:cnd(1,.5,1)-.69146246127401
C:setMode("split screen","full")
```

While entering the command lines, use [F2] [1] to make each line a command line. The first line sets the folder to Main. The second line sets the mode to Approximate, which is how I want to test *cnd().* The next three lines split the screen horizontally, and put the text editor in the top window, and the home screen in the bottom window. This is just personal preference. The next four lines execute my test calls. Note that I subtract the actual desired answer from the *cnd()* result, so I can quickly see if it is working: all the results should be near zero. The last line sets the split screen mode back to Full.

Whenever I want to use this test script, I use [APPS] [8] [2] to open the *cndtest* text file, then [F4] (Execute) to execute each line.

It is not necessary to enter each command line in the text file. Instead, you can use the instructions on page 94 of the *89/92+ Guidebook* to copy most of the commands from the home screen into a text file. However, this won't necessarily work with the split screen commands, as the *setMode()* functions to set the Split 1 application won't work, of course, if you haven't yet created the *cndtest* text file.

Of course, you can automate this testing in another way: by creating a program that does the same operations as the text file.

### [7.26] *dim()* functions slow down with lists and matrices with large elements

There are three 89/92+ functions that return the sizes of lists and matrices: *dim(), rowdim()* and *coldim().* These functions quickly return the dimensions, even for large lists and matrices, as long as the elements in the list or matrix are relatively small. However, if the *elements* are large, these instructions take a long time to return the dimensions. This is true even if the *number* of elements is small. If the elements are too large, a "Memory" error message occurs.

As an example, I wrote a program that created a list with only 70 elements, but the size of the list is about 24K bytes. The elements are high-order polynomials. On my 92+, HW2, AMS 2.04, *dim()* takes over 33 seconds to return the size of this list. If I convert the list to a single-column matrix, *rowdim()* takes over 35 seconds to return the row dimension. If the size of the list increases to 100 elements and about 62K, *dim()* fails and returns a "Memory" error message.

This issue is particularly relevent if you create lists or matrices with large elements, and need to test the dimension in a function that returns a list element, to verify that the matrix index is valid. The whole point of saving large expressions in a list is to be able to quickly return them, instead of recalculating them each time. The purpose is defeated if *dim()* takes 30 seconds to determine if the index is valid.

One potential solution to this dilemma is to use the conditional *try...else...endtry* structure, to access the list element and trap the error if the index is out of range. However, *try...endtry* is not allowed in functions!

### [7.27] *getMode()* returns strings in capital letters

The *setMode()* documentation on page 496 of the *89/92+ User's Guide* shows the *setMode()* arguments as a mixture of upper-case and lower-case letters, for example, the settings strings for "Angle" are "Radian" and "Degree". However, *getMode("angle")* returns the setting in all capital letters, such as "RADIAN" and "DEGREE". This is significant if your program uses *getMode()* to test the current mode setting. This code will never execute either *subrad()* or *subdeg():*

```
if getmode("angle")="radian" then
 subrad()
elseif getmode("angle")="degree" then
```

```
     subdeg()
    endif
```

However, this will work since the test strings are in all capital letters:

```
    if getmode("angle")="RADIAN" then
     subrad()
    elseif getmode("angle")="DEGREE" then
     subdeg()
    endif
```

### [7.28] Use long strings with *setMode()* in custom menus

The Custom menu structure lets you customize the 89/92+ for more efficient operation. You can replace the function key menu tabs at the top of the screen with more useful operations. However, there is a limit to the length of the strings you can use in the Item argument. This limit prevents the use of the long strings needed for some of the *setMode()* function arguments. The solution to this dilemma is to use the number codes for the *setMode()* arguments, instead of the strings themselves.

Suppose we want to make a custom menu to set the Exact/Approx modes to Auto, Exact or Approximate. You might try this program, but it will not work:

```
    custom1()
    Prgm

    custom
     title "Modes"
     item "setmode(""Exact/Approx"",""Auto"")"
     item "setmode(""Exact/Approx"",""Exact"")"               <- FAILS HERE
     item "setmode(""Exact/Approx"",""Approximate"")"
    endcustm

    custmon

    EndPrgm
```

The string for the second Item line, to set the mode to Exact, causes a "Dimension" error because the string is too long. The solution is to create shorter strings by using the number string codes for *setMode(),* instead of the strings themselves. These codes are listed on page 584 of the online *89/92+ Guidebook*.

Note also the use of the double quote marks around the item strings. These are used to embed double quotes in the Item arguments, which are themselves strings.

This is how the program looks using the codes instead of the strings:

```
    custom1()
    Prgm

    custom
     title "Modes"
     item "setmode(""14"",""1"")"
     item "setmode(""14"",""2"")"
     item "setmode(""14"",""3"")"
    endcustm

    custmon
```

```
EndPrgm
```

"14" is the code for Exact/Approx. "1", "2" and "3" are the codes for Auto, Exact and Approximate, respectively. This program works, but you can't really identify tell from the displayed menu which item is which. A solution to this problem is to use extra Item commands, just as labels. With this solution, the program now looks like

```
custom1()
Prgm

custom
 title "Modes"
 item "Set Auto:"
 item "setmode(""14"",""1"")"
 item "Set Exact:"
 item "setmode(""14"",""2"")"
 item "Set Approx:"
 item "setmode(""14"",""3"")"
endcustm

custmon

EndPrgm
```

This program will create a menu for function key F1 that looks like this:

```
1:Set Auto:
2:setmode("14","1")
3:Set Exact:
4:setmode("14","2")
5:Set Approx:
6:setmode("14","3")
```

There are a few problems with this method. First, it makes the menu twice as long as it really needs to be. Second, it is an awkward user interface, because the user's natural inclination is to press [1] for Auto mode, while it is really key [2] that needs to be pressed.

These objections can be overcome by using both codes and strings in the Item arguments. Use the "14" code to shorten strings, but use the strings themselves for Auto, Exact and Approximate:

```
custom1()
Prgm

custom
 title "Modes"
 item "setmode(""14"",""Auto"")"
 item "setmode(""14"",""Exact"")"
 item "setmode(""14"",""Approximate"")"
endcustm

custmon

EndPrgm
```

This program makes a menu that looks like this:

```
1:setmode("14","Auto")
2:setmode("14","Exact")
3:setmode("14","Approximate")
```

If you switch modes often, this type of menu is faster than using the MODE menu. Using the MODE menu takes at least 7 keystrokes. The Custom menu method requires 3 keystrokes. However, the Custom menu method unnecessarily leaves the commands in the history display. You need to press [ENTER] after selecting the menu item, which is just one more keystroke. Finally, the appearance of the menu items is rather poor in that all the *SetMode()* details are displayed. Still, it is an improvement over using the [MODE] key and cursor control keys to make a simple mode change.

*(Credit for use of code strings declined)*

## [7.29] Table of *getConfg()* list elements

The *getConfg()* function can be useful in programs when you need to determine system information. For example, you might want to check the free RAM before creating a large variable, or you may want to check the amount of archive memory available, before your program archives a large variable. This type of information is easily extracted from the list that *getConfg()* returns. For example, if no certificates have been installed, then the free RAM in bytes is returned by `getConfg()[20]`. The table below simply shows the indices for the list, since this information is not explicitly listed in the user's manual.

| Index | No certificates installed: | Certificates installed: |
|---|---|---|
| 1 | "Product Name" | "Product Name" |
| 2 | "Advanced Mathematics Software | "Advanced Mathematics Software |
| 3 | "Version" | "Version" |
| 4 | "2.05, 07/05/2000" | "2.05, 07/05/2000" |
| 5 | "Product ID" | "Product ID" |
| 6 | "01-1-9-4C" | "01-1-9-4C" |
| 7 | "ID #" | "ID #" |
| 8 | "011DC 9FC31 D63A" | "011DC 9FC31 D63A" |
| 9 | "Screen Width" | "Cert. Rev. #" |
| 10 | 240 | 0 |
| 11 | "Screen Height" | "Screen Width" |
| 12 | 128 | 240 |
| 13 | "Window Width" | "Screen Height" |
| 14 | 240 | 128 |
| 15 | "Window Height" | "Window Width" |
| 16 | 91 | 240 |
| 17 | "RAM Size" | "Window Height" |
| 18 | 262,132 | 91 |
| 19 | "Free RAM" | "RAM Size" |
| 20 | 183,044 | 262,132 |
| 21 | "Archive Size" | "Free RAM" |
| 22 | 655,360 | 183,044 |
| 23 | "Free Archive" | "Archive Size" |
| 24 | 583,348 | 655,360 |
| 25 |  | "Free Archive" |
| 26 |  | 583,348 |

Beginning at index 9 the indices differ, depending on whether or not any flash applications have been installed. You can determine if certificates have been installed by testing the dimension of the list returned by *getConfg().* If the dimension is 24, no certificates are installed. As an example, consider this expression which returns the number of free archive bytes, with or without certificates:

```
when(dim(getConfg())=24,getConfg()[24],getConfg()[26])
```

If your program needs one of the first 8 parameters (Product Name through ID#), you need not determine if certificates are installed, since the indices for those parameters are the same in either case. If your programs needs one of the parameters after the Cert. Rev # parameter, you can get it by calculating the index offset from the end of the list. For example, if your program needs the screen width, this expression will return it:

```
getConfg()[dim(getConfg())-14]
```

If no certificate is installed the calculated index is 10; if certificates are present, the index is 12; in either case the index points to the screen width.

### [7.30] *sign()* function is not compatible with other programming languages

Most programming languages have a *sign()* function, that returns 1 if the argument is greater than or equal to zero, and -1 if the argument is less than zero. The 89/92+ *sign()* function *does not* operate like this. Instead, the 89/92+ *sign()* function returns -1 if the argument is less than zero, 1 if the argument is greater than zero, and the expression 'sign(0)' if the argument is zero. While perhaps mathematically rigorous, this is often not the most useful interpretation. This *sgn()* function returns 1 when the argument is zero:

```
sgn(x)
Func
when(x=Ø,1,sign(x))
EndFunc
```

### [7.31] *return* shows variables during debugging

Function debugging is easier if you can see the contents of the local variables. This can be accomplished by inserting a temporary *return* statement in your function. Use *return* with the variables you want to examine. For example, suppose you have a function with a variable *a1*, and you suspect *a1* has the wrong value. Just use

```
...
return a1
...
```

at the point you want to check *a1.* If you want to see more than one variable, use a list:

```
...
return {a1,a2,a3}
...
```

*(Credit to Glenn E. Fisher)*

## [7.32] Documenting programs

Program documentation answers these questions about your programs:

1. What problem does the program solve?
2. How do I use the program?
3. How does the program get its results?
4. How can I change the program to better meet my needs?

Documentation is work and it takes time. For most programmers, documentation is a chore, and not nearly as interesting as designing and coding. If you are just writing programs for yourself, the documentation is obviously your own business. However, if you distribute your programs to others, I assume you really want people to be able to use them. Some useful programs may be so simple that they don't need any documentation, but I have seen any, yet.

The main goal of documentation is to answer the questions above, as quickly as possible for both you and the user. While you may be rightfully quite proud of your program, the user probably does not share your enthusiasm. The user needs to solve a problem, and it looks like your program might do it. Your responsibility in creating the documentation is to meet the user's needs. Those needs can be grouped in two broad categories: answering questions and giving warnings. Some of the questions are obvious, such as: how do I install the program? how do I run it? how do I fix this error? The warnings are less obvious but just as important. You need to inform the user as to side effects that might not be obvious. Some examples of these side effects might be changing the modes, creating folders or creating global variables.

Documentation is just writing. Clear writing is a skill which, like any other, takes practice and commitment. The more you do it, the better (and faster) you will get. In the remainder of this tip, I give some suggestions for creating usable documentation. The suggestions are organized around the four questions that users have.

*What problem does the program solve?*

If you have written a program to solve a problem, you are quite familiar with the problem. It may be a problem in your field of expertise that you solve every day. But other users don't necessarily have your background. They might just be learning about the field. They might be unsure of the terminology and solution techniques. To meet those users' needs, you must have a short, clear, complete description of the problem.

*How do I use the program?*

Your documentation should answer these questions for the user:

- Does the program run on either the 89 or the 92+? Are there special versions for each calculator?
- Does the program require a specific version of the AMS?
- How much ROM does the program take?
- Can the program be archived?
- How do I install the program?
- How do I uninstall the program?

- Does the program need other functions or programs?
- Can the program be intstalled in any folder, or does it have to be in a specific folder?
- How do I start the program?
- How do I exit (quit) the program?
- What Mode settings are needed? Does the program change my Mode settings? Does the program restore my mode settings?
- Does the program change the current folder? Does it restore the previous folder?
- Does the program change my Graph screen? Does it change my Program I/O screen? Does the program change any of my Y= Editor settings?
- Does the program leave any Data plot definitions behind?
- Does the program leave behind any global variables? What are there names? What happens if I delete them, then run the program again?
- What other side effects does this program have?
- How does the program get its inputs from me? What are valid inputs? What are the units for the inputs? Can the inputs be real or complex? If my input is a function name, how do I specify that function?
- If this is a function, how do I call it? What are the calling arguments? Give examples.
- How does the program display the output results? What are the units? Are the results stored anywhere?
- How accurate are calculated results? How can I check the accuracy? Does the accuracy degrade for some input ranges?
- How long does it typically take for the program to get an answer?
- If the program duplicates a built-in function, why is it better? Is it faster, more accurate, or is it more general?
- How much testing did you do on this program?
- Does the program display any error messages? What do the error messages mean? How do I fix the errors?
- Does the program have built-in help? How do I get to it? Can I delete it to save memory?
- Where do I get help, if I need it?
- What is the version of this program?
- If this is a new version of the program, how is it different from the previous version?
- Does this program have any known bugs? How do I avoid them?
- Is this program based on someone else's work? How did you improve on their effort?

This is a fairly complete list, and all of these items might not be appropriate for all programs. You just have to use your judgement as to what best serves the user.

*How does the program get its results?*

This section of the documentation describes *how* the program does what it does. For various reasons, you might not want to tell the user how your program works. I believe this is a disservice to the user. If I am counting on your program to give me a correct answer, I deserve some assurance that it really works. In part, this assurance can come from knowing what algorithms you used to get the results. If the calculation depends on some built-in function, for example, *solve()* or *nint(),* and you tell me that, then I know that your program is limited by the performance of those functions, and I can evaluate the results accordingly.

This section of the documentation can be fairly short. If I really want to know all the intricate details, I can examine your source code. The real purpose of this section is to give an overview.

*How can I change the program to better meet my needs?*

It may be quite likely that your program almost does what the user wants. In this case, the user might want to modify the program. To do this, she needs to understand how the program works, and this means code documentation. Code documentation differs slightly from user documentation, because both the intended audience and the purpose are different. User documentation meets the user's needs, and the user is not necessarily a programmer. Code documentation meets the needs of a programmer. Useful code documentation is actually a field of its own, and I'm only going to give a brief overview of 'best practices', based on my personal experience of programming and supervising programmers. The documentation also interacts to a great degree with the underlying design of the code, which is a topic that deserves a tip of its own.

A program consists essentially of data, calculations, and flow control. Because these three attributes interact with each other, the documentation must specify the interaction, as well as the characteristics of each of attribute. Data documentation consists of

- Choosing a variable name. Ideally, variable names are long and descriptive. On the 89/92+, memory is limited, so there is a strong motivation to keep variable names short, to save memory. In any case, variable names are limited to eight characters, which often precludes truly descriptive names. This means that descriptive documentation of variables is even more important. Some simple conventions can help, as well. For example, integer loop counters are traditionally named with integers *i, j, k, l, m* and *n.* Nested loops can use loop counters such as *k1* and *k2*. Some 89/92+ programmers prefer to use Greek or non-english characters, especially for local variables. Selecting variable names on the 89/92+ is further confounded by the fact that the AMS uses several common, obvious names as system variables and data.
- Describing the data type: is it a variable or a constant? Is it an integer, a string or a matrix? What are the valid ranges and dimensions? What is the variable's *physical* significance? For example, *Vin1* and *Vin2* may be two circuit input voltages, and *Vout* is the output voltage.
- Clarifying variables which change type during the program execution. It is completely legal in TI Basic to store a string to a variable which contains an integer, for example. While this improves efficiency, it can make programs very difficult to understand. Further, it is common (and efficient) to 'reuse' a local variable when the previous contents are no longer needed.

Documenting calculations is fairly straightforward, particularly if the underlying data is well documented. Keep in mind, though, that you want to describe the functional intent of the calculation, not the calculation itself. For example, this isn't too helpful:

```
©Add 273.15 to temp1
temp1+273.15→temp1
```

This is better:

```
©Convert temp1 from degrees C to degrees K
temp1+273.15→temp1
```

Some calculations can be efficiently coded as a single large, complex expression. These are the calculations that really need good explanations. Remember that the purpose of the description is to describe *why* you are doing something, not *what* you are doing - the 'what' is obvious from the code itself.

Documenting the flow control is easy if the program structure is logical to begin with. *goto*'s and labels are discouraged in good structured programming, for good reason, but they can result in the most possible efficient code when used sparingly. It can be challenging to come up with good label names, so I just have a few common names for pervasive elements such as *mainloop* and *exit*. Error handling may interrupt normal program flow, so the program comments should make it clear *where* the error is handled, and *where* control is transfered after the error is handled.

*Where to put the documentation?*

There are at least three possiblities for physically locating the program documentation: in the source code, as a separate *readme* text file, or as an 89/92+ text file. Putting the documentation in the source code file has the advantages that the documentation always travels with the program, but it dramatically increases the code size and uses up valuable on-calculator memory. Programs are most commonly documented with a separate *readme* file. This certainly works, but then the user might not have access to the documentation when they really need it most.

Creating an 89/92+ text file that includes all the program documentation has these advantages:

- The documentation is on the calculator, with the program, and can be readily accessed. If one user transfers the program to another user, by calculator, he can also send the text variable, so the new user has the documentation, too.
- You can use the special 89/92+ characters in the documentation, which is not easily done in a separate *readme* file.
- You can include sample calls in the text file, which can be executed directly from the file.
- The user can edit the text file to meet their needs. For example, when they first start using the program, they may need most of the detailed documentation. As they become more familiar with the program, they can delete the text they don't need to save memory. Finally, they can add their own particular comments as needed.

If you use this method, you should give the name of the documentation text variable in the source code.

### [7.33] Local functions must be declared as local variables

You can define programs and functions within a program in TI Basic. This is convenient, in that very specialized functions are included with the calling program, which simplifies distribution and maintenance. However, those local functions must be declared as local variables with a `Local` statement. This might seem obvious, and it *is* in the manual, but I tend to forget it, anyway.

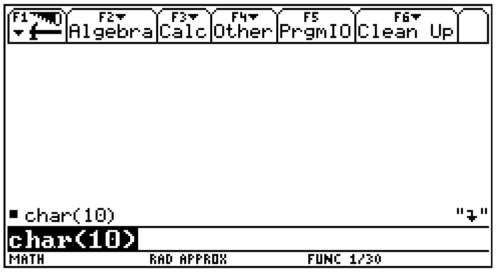
### [7.34] Use linefeed char(10) for separate lines in program Help

Your programs can make use of the Help feature built into the 89/92+ AMS. With this feature, the first comment in the program is shown when you press [F1] Help, in the Catalog display. You can embed line feed characters in the single comment line, to format the comments for better appearance and readability.

To create the linefeed character, evaluate char(10) on the command line:

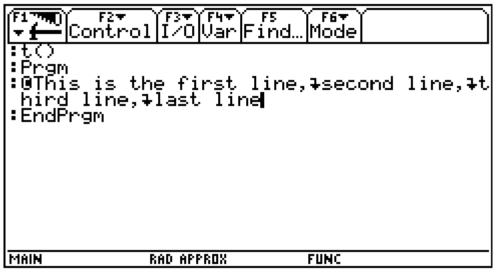
```
char(10) [ENTER]
```

which puts the linefeed character in a string in the history display:



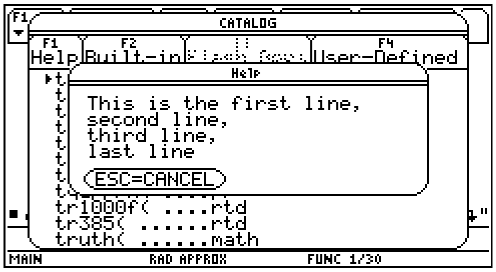
```
■ char(10)                              "↵"
char(10)
MATH          RAD APPROX        FUNC 1/30
```

Use [UP] to highlight the string in the history area, then [ENTER] to paste it into the command line. Use [SHIFT] and the cursor keys to highlight just the linefeed character (not the double quote marks), then copy it. Open the program you want to comment, and use [PASTE] to insert the linefeed character where needed. This shows an example:



```
:t()
:Prgm
:©This is the first line,↵second line,↵t
 hird line,↵last line
:EndPrgm
MAIN          RAD APPROX        FUNC
```

Then the Help screen looks like this:



```
This is the first line,
second line,
third line,
last line
```

```
tr1000f( ....rtd
tr385( ......rtd
truth( ......math
MAIN          RAD APPROX        FUNC 1/30
```

You can paste multiple line feed characters to insert blank lines between the comments.

This method fails if you open and save the program in GraphLink: the linefeed characters are removed. There is no known way to insert the linefeed characters in GraphLink.

*(Credit to Bhuvanesh Bhatt)*

### [7.35] Conditonal test in *If ...EndIf* evaluates all 'and' arguments

Combining some arguments with the Boolean *and* operator will cause a TI Basic program to fail. For example:

```
If GetType(lista)="LIST" and lista[1]=2 then
 3→lista[1]
EndIf
```

In this example a program error occurs if the variable *lista* is not actually a list. This problem is avoided by using two nested *If* statements:

```
If GetType(lista)="LIST" then
 If lista[1]=2 then
  3→lista[1]
 EndIf
Endif
```

*{Behavior discovered by Kosh DV}*

### [7.36] Input arguments for 'keyboard' programs

The TI-89 / TI-92 Plus support up to nine 'keyboard' programs. These programs can be executed from any folder by pressing [DIAMOND][1] for *kbdprgm1(),* [DIAMOND][2] for *kbdprgm2(),* and so on. Unlike other programs, keyboard programs cannot have input arguments. However, keyboard programs *can* use the *ans*() and *entry*() functions. *ans*(n) is a built-in function which returns answer *n* from the history display, and *entry*(n) returns entry *n*. These functions can be used to provide input to keyboard programs. As a simple example, suppose we want *kbdprgm2()* to convert a temperature to Kelvin degrees. The program is

```
kbdprgm2()
Prgm
dialog
 text string(tmpcnv(expr("ans(1)"),_°K))
enddlog
EndPrgm
```

Note that we cannot just use *ans(1)* as the *tmpCnv()* argument.  If we did, the TI Basic tokenizer would permanently replace *ans(1)*, in the program, with the actual value of *ans(1)* at the time the program ran. Using the expression *expr("ans(1)")* prevents this silly behavior.

To use the program, enter the temperature to convert, then press [DIAMOND][2], and a dialog box shows the converted temperature. For example, to convert 32°F to K, press

32 [_] [°] [f] [ENTER]

to put 32_°F in the history display, then press [DIAMOND] [2] to run *kbdprgm2()*. A dialog box shows the result of 273.15K.

For conversion programs such as this, it would be even more useful if the result is returned to the history display, as with functions. This is not possible using built-in TI Basic functions, but it *can* be done with the *copyto_h()* ASM program described in tip [7.8]. This version of *kbdprgm2()* uses *copyto_h()* to return the result to the history display:

```
kbdprgm3()
Prgm
Local t,tk
expr("ans(1)")→t
tmpCnv(t,_°K)→tk
util\copyto_h(tmpcnv("t","tk")
EndPrgm
```

As before, the temperature is entered in the history display. When [DIAMOND][3] is pressed to run *kbdprgm3()*, the result is directly returned to the history display.

*(Credit to Andrew)*

### [7.37] Simulate SWITCH or CASE statement with GOTO and indirection

Some computer languages include a SWITCH or CASE statement which transfers program control to a label based on the value of an integer variable. For example, some versions of Basic implement this structure as an ON ... GOTO statement:

```
ON i GOTO l1,l2,l3
```

Program control is transferred to l1, l2 or l3, when i=1, 2 or 3, respectively.

TI-Basic does not have this statement built-in, but it can be simulated using *goto*, indirection and string manipulation.

*First method*

As an example, suppose we have a variable *i* and we want to branch to labels l1, l2 or l3 when *i* is 1, 2 or 3, respectively. The branch is accomplished with

```
goto #(mid("l1l2l3",2*i-1,2))
```

The program labels are specified as a single string, and the *mid()* function extracts the two label characters based on the value of *i*. Note that all the labels must have the same number of characters (two in this example), and *i* must be a valid value, or an error will result. The error can be avoided and handled by using the *try...endtry* structure.

In general, if each label is *n* characters long, then the general form of the expression is

```
goto #(mid(label_string,n*i-(n-1),n))
```

For example, if each label has four characters, then the expression is

```
goto #(mid(label_string,4*i-3,4))
```

*2nd method*

This method needs no string of all the labels, and directly builds the label from the index *i*. The general form is

```
    goto #(label_root&string(exact(i)))
```

where *label_root* is the 'root' of the label names, and *i* is the index. For example, if the labels are l1, l2 and l3, then the root is "l" and the expression is

```
    goto #("l"&string(exact(i)))
```

The *exact()* function is used to convert the index to an integer string with no decimal points; see tip [8.1] for details. As with the first method, *try...endtry* can be used to trap errors from invalid index values.

*(Credit to Samuel Stearly (first method) and Stuart Dawson (second method))*

## [7.38] Why use TI Basic?

TI Basic is the built-in programming language of the TI-89 and TI-92 Plus calculators. It is one of several programming languages you can use; the other languages are M68000 assembly language and two C compilers. "TI Basic" is not Texas Instruments' name for the built-in programming language. In fact, the TI-92 FAQ says this about it:

> ***"Programming language of the TI-92 - is it BASIC?***
> *No. There are a number of features that are similar to the BASIC programming language, but it is not BASIC."*

The *Getting Started* web page for the TI-89/92+ SDK has the only TI reference to "TI Basic" I have found, but I will use that term since it is common and well-understood in the calculator community.

Some programmers claim that TI Basic is unsuitable for coding, citing real and imagined advantages of C. TI Basic does have some serious limitations, but there remain several compelling reasons to use it:

- TI Basic is built into every calculator, and programs can be completely developed on the calculator. You don't need an external interpreter, assembler, compiler, editor or development system.
- The TI Graph Link software can be used for program development if desired. Programs can be edited on the PC, then downloaded to the calculator for execution and debugging.
- The learning curve for TI Basic is short and shallow compared to learning C or assembler.
- TI Basic is stable and robust. It is either difficult or impossible to crash the calculator with a TI Basic program.
- There is no 24K ASM limit for TI Basic programs. You need no hacks or patches to run large TI Basic programs.
- TI Basic functions may return results to other functions or programs, or to the home screen.
- TI Basic functions may be used in expressions, just like built-in functions. This can be done with C functions, but it requires a patch and a hack with AMS 2.05 and HW2 calculators.

- TI Basic can be used to add functional extensions to C or ASM programs which have reached the 24K ASM limit.
- TI Basic is extremely well documented. The user's guide thoroughly describes the language and provides many examples. The user's guide is available in many languages as both a printed document and an Acrobat PDF file. If this isn't enough, there are a few web tutorials to help you get started.
- TI Basic is 'fast enough' for many real applications. This is certainly the case when the application speed is bound by CAS or numeric operations, since both C and TI Basic use the same system calls to perform these functions.
- TI Basic is reasonably complete and sophisticated.
- TI Basic is popular, so there is a large library of existing functions and programs you can use.
- TI Basic is similar to other Basic dialects, so a huge code resource is available in books, magazines and web archives. It is trivial to translate programs from any version of Basic to TI Basic.
- Because of its popularity you can easily get help by posting on the TI discussion groups.
- TI Basic can be extended with assembler and C programs, to fill some of the more glaring functional voids.

In spite of these advantages, there are several areas in which TI Basic could stand some improvement. Loops are extremely slow. Debugging facilities are essentially nonexistent; programs cannot be single-stepped and there are no breakpoints and no facilities for watching variables. TI Basic lacks system function calls, although these can be accomplished with ASM program extensions.

Eric S. Raymond, in *The Jargon Lexicon,* version 4.3.1, gives this definition for BASIC:

> *BASIC /bay'-sic/ n.*
>
> *A programming language, originally designed for Dartmouth's experimental timesharing system in the early 1960s, which for many years was the leading cause of brain damage in proto-hackers. Edsger W. Dijkstra observed in "Selected Writings on Computing: A Personal Perspective" that "It is practically impossible to teach good programming style to students that have had prior exposure to BASIC: as potential programmers they are mentally mutilated beyond hope of regeneration." This is another case (like Pascal) of the cascading lossage that happens when a language deliberately designed as an educational toy gets taken too seriously. A novice can write short BASIC programs (on the order of 10-20 lines) very easily; writing anything longer (a) is very painful, and (b) encourages bad habits that will make it harder to use more powerful languages well. This wouldn't be so bad if historical accidents hadn't made BASIC so common on low-end micros in the 1980s. As it is, it probably ruined tens of thousands of potential wizards.*
>
> *[1995: Some languages called `BASIC' aren't quite this nasty any more, having acquired Pascal- and C-like procedures and control structures and shed their line numbers. --ESR]*
>
> *BASIC stands for "Beginner's All-purpose Symbolic Instruction Code". Earlier versions of this entry claiming this was a later backronym were incorrect.*

Mr. Raymond finds it necessary (or at least amusing) to include Edsger Dijkstra's hoary quote with the memorable phrase 'mentally mutilated'. Even so, it is possible to write clear, coherent maintainable code in BASIC, just as it is possible to code a convoluted mess in C++ or Perl. Regardless, TI Basic *is* what we have on the calculator, so it may be more productive to make the best of the situation rather than moan about its deficiencies.

While TI Basic is particularly suited for quick, short programs and functions, it is certainly possible to use it to code significant applications: Roberto Perez-Franco's symbolic circuit simulator *Symbulator* is one good example. Applications which require maximum speed and fine control of the keyboard and display, as well as access to system functions, are best coded in C.

If you want to learn calculator programming, TI Basic is a good place to start, comments of Raymond and Dijkstra notwithstanding. You will get working results faster than trying to learn C if you have never programmed before. You may find that TI Basic is all you need. If not, it will be easier to learn C once you have some TI Basic experience.

### [7.39] Quickly delete locked, archived variables

Two steps are required to delete an archived variable: first you unarchive the variable, then delete it. If the variable is also locked, three steps are required. These steps are necessary whether you use the functions *unlock()* and *unarchiv()*, or use the VAR-LINK screen. This simple program deletes any variable or list of variables, regardless of whether they are archived or locked.

```
delvar1(n)
Prgm
©("v"), ({"v",...}) Delete v locked/archived
©26dec01/dburkett@infinet.com
local k                              © Loop counter

If getType(n)="STR"                  © Convert single argument to list
{n}→n

If getType(n)="LIST" then            © Delete list of variables
for k,1,dim(n)
  if getType(#(n[k]))≠"NONE" then    © Ignore nonexistent variables
   try:unarchiv #(n[k]):else:endtry  © Must unarchive before unlocking
   try:unlock #(n[k]):else:endtry
   delvar #(n[k])
  endif
 endfor
else                                 © Display argument type error
dialog
  title "DELVAR1"
  text "Bad argument type"
 endDlog
endif

EndPrgm
```

The variable name argument is a string or list of strings. To delete a single variable *var*, use

```
delvar1("var")
```

To delete more than one variable, use a list:

```
delvar1({"var1","var2","var3"})
```

A single string argument is first converted to a list, so either argument type is handled consistently, which saves some program memory. Variables which do not exist are ignored. This may not be the most appropriate behavior, but at least it is consistent with the behavior of *unarchiv()*, *unlock()* and *delvar()*. An error message is displayed in a dialog box if the argument is neither a string nor a list.

This program deletes variables which are locked, archived or both. The order of the *unarchiv* and *unlock* commands is important, because a variable must be unarchived before it is unlocked, or a "Memory" error occurs. This program works because no errors occur if you unlock or unarchive a variable which is already unlocked or unarchived. This means that the program can also be used as a quick way to delete a list of variables which are not locked or archived.

The *unarchiv* and *unlock* commands are used in *Try ... EndTry* blocks, otherwise an error occurs if you try to delete system variables. Many system variables can be deleted, and it is often useful to do so.

This program should obviously be used with some care, since there are no prompts to verify that you really want to delete the variables.

### [7.40] Recall expression without variable value substitution

You can use the RCL operation ([2nd] [STO] on the TI-89 and TI-92+) to recall an expression without substituting variable values. For example, suppose you store an equation to *eqn1*:

```
a+b=c→eqn1
```

then if you evaluate *eqn1* at the entry line, the result a+b=c is returned as expected. However, if you store numeric values to 1, 2, and 3 to *a*, *b* and *c*, respectively, then evaluating *eqn1* returns *true*, since 1+2 = 3. You can recall the original expression and avoid the variable value substitution with [RCL].

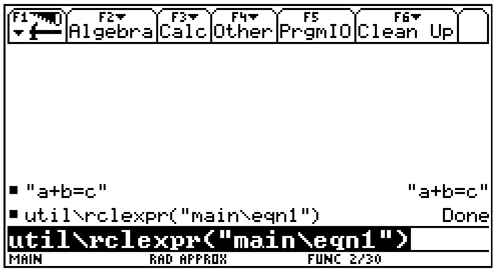
This operation can be accomplished in a TI Basic program, but the RCL operation is not programmable. Instead, we need to use a different method, as this code example shows.

```
rclexpr(ë)
Prgm
local ö,ü

Try:newFold(ä):else:endTry    © Create new folder if necessary
setFold(ä)→ö                  © Make new folder current and save old folder
#ë→ü                          © Evaluate expression
setFold(#ö)                   © Restore old folder
util\copyto_h("ü")            © Return expression as a string

EndPrgm
```

The expression is evaluated in a folder in which the variables do not exist, so substitution cannot take place. The expression is returned as a string, otherwise the variable values would be substituted. The argument ë is a string which specifies the variable and its folder. The folder must be specified, since the expression is evaluated in a different folder. As an example, suppose that the variable *eqn1* and the variables *a*, *b* and *c* are all in the *main*\ folder. If *rclexpr()* is in the *util*\ folder, then the call to get the expression in *eqn1* is

```
util\rclexpr("main\eqn1")
```

and the history display looks like this after execution:



```
■ "a+b=c"                            "a+b=c"
■ util\rclexpr("main\eqn1")             Done
util\rclexpr("main\eqn1")
MAIN          RAD APPROX       FUNC 2/30
```

Note that the expression is returned in the second history line, above the call to *rclexpr()*.

The *copyto_h()* function is used to send the result to the history area; see tip [7.8], *Copy program results to home screen*. The expression is evaluated in the folder *ä*, which is an empty folder containing no variables. *rclexpr()* creates this folder with *newFold()* if it does not already exist. *newFold()* is a block in a *Try ... endTry* structure, which prevents errors if folder *ä* already exists.

*rclexpr()* is useful in its own right, and the general idea can also be used in your own programs, but not in functions, because *Try ... endTry* is not allowed in functions.

### [7.41] Indirection bug with local variables

Indirection fails with local variables, both in the *Local* statement list and when passed as a function or program argument. This program demonstrates the bug:

```
testtype(n)
Func
Local t
getType(#n)→t
Return t
EndFunc
```

Note that *n* is a local variable since it is the argument, and *t* is declared as a local variable. For these following initialized variables:

```
"s"→test2\m           123→test1\m           123→test1\n           123→test1\t
```

*testtype()* gives these results:

```
testtype("test2\m")   correctly returns "STR"
testtype("test1\m")   correctly returns "NUM"
testtype("test1\n")   returns "NONE" in error, since test1\n is a global variable
testtype("test1\t")   returns "NONE" in error, since test1\t is a global variable
```

One work-around is to use special characters for local variable names, for example:

```
ä, ë, ï, ö, ü, ÿ, ä1, äë1, kä
```

With this method, you are assuming that the calculator user will have global variables with these names. You can combine special characters with conventional characters for program readability, and to further reduce the chance that the user will have a global variable of the same name. For example, *testtype()* can be coded as shown, and returns correct results:

```
testtype(än)
Func
```

```
Local ät
getType(#än)→ät
Return ät
EndFunc
```

A similar method embeds underscore characters "_" in the variable names. For example, you could use *n_* for *n*, *t_* for *t*, and so on. Again, you assume that the user does not use these names.

As another work-around, *expr()* can be used to build expressions to evaluate, for example

```
v→#n            becomes       expr("v→"&n)
#n→v            becomes       expr(n&"→v")
getType(#n)→n   becomes       expr("getType"&n&")→t")
```

With this method, this version of *testtype()* returns the correct results:

```
testtype(n)
Func
Local t
expr("getType("&n&")→t")
Return t
EndFunc
```

*(Credit to Martin Daveluy)*

### [7.42] Find variable names used in expressions

It is often useful to extract the variable names from an expression or equation. For example, the variables can be deleted, archived, initialized and so on. Timité Hassan and Bhuvanesh Bhatt, have both written functions to extract variable names. This tip discusses both methods.

*Timité's TI Basic version*

Timité's function *getvars*() returns the variable names of an expression passed as a string. I have slightly modified his function, and changed the name to *exprvars()* to avoid confusion. My modifications change the external subroutines to local routines, and convert some nested-If expressions to *when()* functions. The basic algorithm and code remains unchanged.

```
exprvars(g_str)
Func
©("expr") return list ofvariables in "expr"
©Timité Hassan, modified by dab

Local k,i,cc,g_vr,g_d,g_dim,g_ll,ss,varin,vardeb

Define varin(c)=Func
Local cc
ord(c)→cc
when(c≥"a" and c≤"z",true,when(c≥"0" and c≤"9",true,when(cc≥128 and cc≤148 and
cc≠140,true,when(c="_",true,false))))
EndFunc

Define vardeb(c)=Func
Local cc
ord(c)→cc
when(c≥"a" and c≤"z",true,when(cc≥128 and cc≤148 and cc≠140,true,false))
EndFunc

dim(g_str)→g_dim
```

```
{}→g_ll
Ø→g_d
For k,1,g_dim
 mid(g_str,k,1)→cc
 if cc≥"A" and cc≤"Z":char(ord(cc)+32)→cc
 If vardeb(cc) then
  cc→ss
  k+1→k
  mid(g_str,k,1)→cc
  if cc≥"A" and cc≤"Z":char(ord(cc)+32)→cc
  while k≤g_dim and varin(cc)
   ss&cc→ss
   k+1→k
   mid(g_str,k,1)→cc
   if cc≥"A" and cc≤"Z":char(ord(cc)+32)→cc
  Endwhile
  If cc≠"(" and ss≠"and" and ss≠"or" and ss≠"xor" then
   For i,1,g_d
    if ss=g_ll[i]:exit
   Endfor
   If i>g_d then
    augment(g_ll,{ss})→g_ll
    g_d+1→g_d
   Endif
  Endif
 Endif
Endfor
g_ll
Endfunc
```

Note that the expression is passed as a string, and need not be a valid expression. The variable names are returned as string elements of a list. For example, the call

```
exprvars("a+b/c+sin(d^e)+22.2=f")
```

returns

```
{"a","b","c","d","e","f"}
```

*Timité Hassan's original getvars() function*

This is Timité's original *getvar()* code. I include it here out of courtesy, and so that if I have made in mistake in my changes, you can use the original code. *getvars()* calls *varin()* and *vardeb()*.

*getvars():*

```
getvars(g_str)
Func
Local k,i,cc,g_vr,g_d,g_dim,g_ll,ss

dim(g_str)→g_dim
{}→g_ll
Ø→g_d
For k,1,g_dim
mid(g_str,k,1)→cc
if cc≥"A" and cc≤"Z"
char(ord(cc)+32)→cc
If vardeb(cc) then
cc→ss
k+1→k
mid(g_str,k,1)→cc
if cc≥"A" and cc≤"Z"
char(ord(cc)+32)→cc
while k≤g_dim and varin(cc)
```

```
ss&cc→ss
k+1→k
mid(g_str,k,1)→cc
if cc≥"A" and cc≤"Z"
char(ord(cc)+32)→cc
Endwhile
If cc≠"(" and ss≠"and" and ss≠"or" and ss≠"xor" then
For i,1,g_d
if ss=g_ll[i]
exit
Endfor
If i>g_d then
augment(g_ll,{ss})→g_ll
g_d+1→g_d
Endif
Endif
Endif
Endfor
g_ll
Endfunc
```

*vardeb():*

```
vardeb(c)
Func
Local cc
ord(c)→cc
If c≥"a" and c≤"z" then
 Return true
Elseif cc≥128 and cc≤148 and cc≠140 then
 Return true
Endif
False
EndFunc
```

*varin():*

```
varin(c)
Func
Local cc
ord(c)→cc
If c≥"a" and c≤"z" then
 Return true
Elseif c≥"0" and c≤"9" then
 Return true
Elseif cc≥128 and cc≤148 and cc≠140 then
 Return true
Elseif c="_" then
 Return true
Endif
False
EndFunc
```

*Bhuvanesh' C version: varlist()*

Bhuvanesh' version of this function is called *VarList()*, and differs functionally from *exprvars()* in that the input argument can be passed as a string or an expression, and an optional input argument excludes user functions from the variable list.

```
VarList(expr|string,{excludeFunctions})
```

where *expr* | *string* is the expression or string. Set *excludeFunctions* to 1 to exclude user functions from the list, or omit it to include user functions. *VarList()* must be executed in Auto or Exact mode.

Some examples:

| | | | |
|---|---|---|---|
| `VarList(a+b*x)` | returns | {a, b, x} | |
| `VarList("a+b")` | returns | {a, b} | |
| `VarList(f(x,y))` | returns | {y, x, f} | *(function name included*) |
| `VarList(f(x,y),1)` | returns | {y, x} | *(function name excluded*) |

### [7.43] Faster function calls with list and matrix arguments

Function call execution time increases substantially with list and matrix arguments. The increase is independent of the code executed by the function, so I call it *overhead*. This tip quantifies the overhead and shows a method to reduce it, at least for global lists and matrices.

I use various timing results in this tip. All execution time data is for a HW2 TI-92 Plus, AMS 2.05. HW1 calculators run about 17% slower. I tested lists and matrices with random floating point integer elements, so execution times will be different if the elements are integers or symbolic elements.

#### ***Matrix arguments***

I used this test program to measure the overhead for a HW2 TI-92 Plus, AMS 2.05:

```
t()
Prgm
Local k,s,res

Define s(mat)=Func     © Define a function with a single matrix argument
 Return 0              © Just return a constant
EndFunc

For k,1,50             © Loop fifty times for better timing resolution, ...
 s(m)→res              © ... calling the subroutine with the matrix argument
EndFor

EndPrgm
```

*m* is a global matrix, initialized to random integer values with *randmat()*. Tests with 34 matrices, with up to 200 rows and 20 columns, resulted in this model equation to estimate the execution time:

$$T = a + b \cdot n_r + c \cdot n_c + d \cdot n_c \cdot n_r \qquad [1]$$

where *T* is the execution time for each call, and

$n_r$ = number of matrix rows
$n_c$ = number of matrix columns

a = 2.5404 E-2　　b = 8.2538 E-4　　c = 7.615 E-5　　d = 8.2447 E-4

This model equation is accurate to about 3%. This table shows some typical overhead times:

| Number of rows<br>nr | Number of columns<br>nc | Execution time<br>per call |
|---|---|---|
| 10 | 1 | 42.0 mS |
| 10 | 4 | 66.9 mS |
| 100 | 1 | 191 mS |
| 100 | 4 | 438 mS |
| 200 | 1 | 355 mS |
| 200 | 3 | 685 mS |
| 3 | 200 | 538 mS |
| 5 | 5 | 50.5 mS |
| 0 | 10 | 117 mS |
| 20 | 20 | 373 mS |

The overhead is about 0.7 seconds for the largest tested matrix. This may not be critical for single function calls, but it is significant if the function is called repeatedly, as is the case when using the numeric solver, performing numeric integration, or plotting the function. For example, if the 200 x 3 matrix is passed to a function which is plotted at each of the 240 display pixel columns of the TI-92 Plus, the overhead is about 2.7 minutes.

If more than one matrix is used as a function argument, the overhead predictably increases to about the sum of the individual overheads of each matrix. For example, if the arguments are two matrices with dimensions of 50 rows x 2 columns, and 50 rows x 3 columns, the total overhead is about 149 mS + 191 mS = 340 mS.

The overhead may be reduced by passing the matrix by reference, instead of by value. For the timing results and model equation above, the matrix was passed by value, that is, the matrix itself was passed as the function argument. A typical function call is

`f(matrix)` *(call by value)*

To pass the matrix by reference instead of value, the function argument is the matrix name as a string:

`f("mat_name")` *(call by reference)*

With the value method, the element [2,3] is accessed with

`mat[2,3]` *(access by value)*

With the reference method, the same element is accessed using indirection:

`#mat_name[2,3]` *(access by reference)*

where *mat_name* is the name of the matrix. Indirection, a feature built into TI Basic, is accomplished when the number character '#' precedes a variable name, for example, #*var*. This expression does not return the value of *var*, instead, it returns the value of the variable whose name is stored as a string in *var*. For example, if *var* contains the string "var2", and the value of variable *var2* is 7, then #var returns 7.

The reference method works only when the matrix is a global variable. A variable which is local in a program or function cannot be accessed by any other function which is called.

These two test routines demonstrate the overhead reduction when using the reference method.

```
tval()
Prgm
© Pass matrix by value

Local f1,k,r

Define f1(mat)=Func
 Return mat[1,1]
EndFunc

For k,1,50
 f1(m)→r
EndFor

EndPrgm
```

```
tref()
Prgm
© Pass matrix by reference (name)

Local f2,k,r

Define f2(mat)=Func
 Return #mat[1,1]
EndFunc

For k,1,50
 f2("m")→r
EndFor

EndPrgm
```

The *tval()* routine passes the matrix *m* by value, and *tref()* passes the matrix name as a string. *tval()* takes about 199 mS for each function call, while *tref()* takes about 31 mS. This is an improvement of about 84%.

Even though it is faster to call a function with the reference method, it takes longer to access the matrix elements with indirection. For some combination of function calls and matrix element accesses the total execution time is the same for either method. This is called the break-even point. In general, the total execution time is

$$T = N_c T_c + N_a T_a$$

where *Nc* and *Na* are respectively the number of function calls and element accesses, and *Tc* and *Ta* are the execution times for each function call and element access. More specifically, the total execution times for each method are

$T_v = N_{cv} T_{cv} + N_{av} T_{av}$ *Value method*

$T_r = N_{cr} T_{cr} + N_{ar} T_{ar}$ *Reference method*

The break-even point is at Tv = Tr. We want to compare the two methods with the same conditions, so we equate the number of function calls and element accesses for each method:

$$N_{cv} = N_{cr} = N_c \qquad N_{av} = N_{ar} = N_a$$

Equate the expressions for Tv and Tr: $N_c T_{cv} + N_a T_{av} = N_c T_{cr} + N_a T_{av}$

and solve for Nc: $N_c = N_a \frac{T_{ar} - T_{av}}{T_{cv} - T_{cr}}$ or $\frac{N_c}{N_a} = \frac{T_{ar} - T_{av}}{T_{cv} - T_{cr}}$ [2]

With equation [2], we can find the break-even point at which the two methods have the same execution time, for some number of function calls and matrix accesses. Tcv is found with equation [1] above, and Tcr is a constant:

$$T_{cr} = 16.88 \text{ mS/access}$$

Timing experiments show that the time required to access a matrix element depends on the number of matrix rows and columns, as well as the element's location in the matrix. The access time also changes slightly if the indices are constants or variables. Timing data shows that Tar - Tav is relatively constant at about 5.7 mS, so equation [2] simplifies to

$$\frac{Nc}{Na} = \frac{5.7\ mS}{Tcv - 16.88\ mS} \qquad [3]$$

For example, suppose we have a matrix with 50 rows and 3 columns. We use equation [1] to find Tcv = 191 mS, and Nc/Na = 0.033. If the function accesses only three elements (Na=3), then Nc = 0.1. Since Nc < 1, the reference method is always faster, regardless of the number of function calls. However, if the function accesses every matrix element (Na = 150), then Nc = 4.95. In this case, the reference method is faster only if we call the function more than 5 times.

As another example, consider a matrix with 10 rows and 10 columns. Equation [1] gives Tcv = 117 mS, and Nc/Na = .057. If the function accesses 10 elements, then Nc = .57, so the reference method is always faster. However, if the function accesses all 100 matrix elements, then Nc = 5.7, so the execution time is less for the reference method only if we call the function at least 6 times.

The general result is that the reference method may be faster if the matrix is large and few elements are accessed.

### *List arguments*

When lists are passed as function arguments, the call overhead can also be considerable, as this table shows:

| Number of list elements | Overhead / function call |
|---|---|
| 0 | 22 mS |
| 100 | 84.6 mS |
| 400 | 269.1 mS |
| 700 | 459.8 mS |

Timing data for lists of 11 sizes results in this model equation, which is accurate to 1 or 2%:

T = 623.2967E-6 (N) + 0.021577 [4]

where *T* is the overhead per function call and *N* is the number of list elements. As with the case for matrices above, we can find a function to calculate the break-even point in terms of the number of function calls, and the number of list element accesses made by the function. It is, in fact, the same function with different constants, that is

$$\frac{N_c}{N_a} = \frac{T_{ar} - T_{av}}{T_{cv} - T_{cr}}$$

where

Nc = the number of function calls
Na = the number of list element accesses in the function

Tar = time required to access an element by reference: *expr(list_name&"[]")*
Tav = time required to access an element by value: *list[]*
Tcv = time required to call the function with the list as a value argument; from [4] above
Tcr = time required to call the function with the list argument by name

Tar and Tav are functions of the size of the list, as well as which element of the list is accessed. These functions estimate the mean execution time for a single list element:

$$T_{ar}(N) = 23.0918E - 6 \cdot N + 16.948E - 3$$

$$Tav(N) = 23.1404E - 6 \cdot N + 10.368E - 3$$

where *N* is the number of list elements, so

$$T_{ar}(N) - T_{av}(N) = 48.6E - 9 \cdot N + 6.58E - 3$$

The small *N* coefficient can be ignored. More timing experiments give

Tcr = 16.9 mS

So the equation simplifies to $$\frac{N_c}{N_a} = \frac{6.58\ mS}{T_{cv} - 16.9\ mS} \qquad [5]$$

For example, for a list of 100 elements, equation [4] gives Tcv = 83.91 mS. If three elements are accessed each function call (Na = 3), then Nc is 0.29, which means that the reference method is always faster. If all 100 elements are accessed each call (Na = 100), then Nc = 9.8, which means that the reference method will be faster when the function is called at least 10 times.

I mentioned above that the element access time also depends on the size of the list. For a list consisting of 250 elements, it takes about 16 mS to access element [1], 22 mS for element [125], and 28 mS for element [250]. This is substantial variation, which means that the mean timing results above apply only if the probabilities of accessing any particular elements are equal. If you know that your application tends to access elements at a particular list index, you need to account for that in the break-even analysis.

### *Execution time data for matrix element accesses*

This section shows the test data for matrix element access times. All times are for a TI-92 Plus, HW2, AMS 2.05. The test matrices are created with randmat(), so the matrix elements are floating point integers. The mode settings for the tests are RAD and APPROX.

Table 1 shows the time required to access a single matrix element when the matrix indices are constants, for example, *matrix[1,1]*. The test data shows that the overhead to access a matrix element with the reference method (by indirection) is about 5.3 mS. The data also shows the access time variation with respect to the size of the matrix, and the element's position in the matrix.

***Table 1***
***Matrix element access time, constant indices***

| **Matrix dimensions** | **Element accessed** | **Access time Tav, value method (mS)** | **Access time Tar, reference method (mS)** | **Reference method overhead Tar - Tav** |
|---|---|---|---|---|
| 10 rows, 10 columns | [1,1] | 14.3 | 19.2 | 4.9 |
| | [5,5] | 16.7 | 21.8 | 5.1 |
| | [10,10] | 19.3 | 24.5 | 5.2 |
| | | | | |
| 20 rows, 20 columns | [1,1] | 14.2 | 19.4 | 5.2 |
| | [5,5] | 17.7 | 23.4 | 5.7 |
| | [10,10] | 22.8 | 28.2 | 5.4 |

| | | | | |
|---|---|---|---|---|
| | [15,15] | 27.8 | 33.3 | 5.5 |
| | [20,20] | 32.7 | 38.2 | 5.5 |

Table 2 shows the time required to access a matrix element when the indices are local variables, for example, *matrix[m1,m2]*. The data shows that it takes slightly longer to access an element with variable indices, but the overhead for indirect access is about the same, at a mean of 5.8 mS.

***Table 2***
***Matrix element access time, variable indices***

| **Matrix dimensions** | **Element accessed** | **Access time Tav, value method (mS)** | **Access time Tar, reference method (mS)** | **Reference method overhead Tar - Tav** |
|---|---|---|---|---|
| 10 rows, 10 columns | [1,1] | 15.5 | 21.2 | 5.7 |
| | [5,5] | 17.9 | 23.7 | 5.8 |
| | [10,10] | 20.6 | 26.2 | 5.6 |
| | | | | |
| 20 rows, 20 columns | [1,1] | 15.5 | 21.3 | 5.8 |
| | [5,5] | 19.5 | 25.2 | 5.7 |
| | [10,10] | 24.2 | 30.2 | 6.0 |
| | [15,15] | 29.5 | 35.3 | 5.8 |
| | [20,20] | 34.2 | 40.2 | 6.0 |

Table 3 shows the mean time required to access each element of matrices of various sizes. Element indices are local variables, and the elements were accessed by value, for example, *matrix[m1,m2]*.

***Table 3***
***Mean matrix element access time, direct access***

| **Matrix rows** | **Matrix columns** | **Mean access time, single element (mS)** |
|---|---|---|
| 5 | 25 | 18.9 |
| 5 | 35 | 19.7 |
| 5 | 45 | 20.9 |
| 5 | 55 | 21.4 |
| 5 | 65 | 22.5 |
| 5 | 75 | 23.6 |
| 5 | 85 | 25.0 |
| 5 | 95 | 25.9 |
| 5 | 105 | 27.0 |
| 5 | 115 | 27.9 |
| 5 | 125 | 28.9 |
| 5 | 135 | 30.1 |
| 5 | 145 | 31.3 |
| 5 | 155 | 32.4 |
| 10 | 10 | 18.2 |
| 10 | 20 | 20.3 |
| 10 | 30 | 22.6 |
| 10 | 40 | 24.9 |

| **Matrix rows** | **Matrix columns** | **Mean access time, single element (mS)** |
|---|---|---|
| 20 | 10 | 21.5 |
| 20 | 20 | 25.5 |
| 20 | 30 | 29.3 |
| 25 | 5 | 19.1 |
| 30 | 10 | 23.8 |
| 30 | 20 | 30.0 |
| 35 | 5 | 21.2 |
| 40 | 10 | 25.2 |
| 40 | 20 | 34.0 |
| 45 | 5 | 21.9 |
| 50 | 10 | 27.7 |
| 55 | 5 | 23.6 |
| 60 | 10 | 30.3 |
| 65 | 5 | 25.0 |
| 70 | 10 | 32.9 |
| 75 | 5 | 26.0 |
| 80 | 10 | 35.4 |
| 85 | 5 | 27.4 |

| | | |
|---|---|---|
| 10 | 50 | 26.8 |
| 10 | 60 | 30.9 |
| 10 | 70 | 31.0 |
| 10 | 80 | 33.3 |
| 15 | 15 | 20.5 |
| 15 | 25 | 23.7 |
| 15 | 35 | 26.9 |
| 15 | 45 | 30.3 |
| 95 | 5 | 29.1 |
| 105 | 5 | 30.5 |
| 115 | 5 | 31.7 |
| 125 | 5 | 33.2 |
| 135 | 5 | 34.9 |
| 145 | 5 | 36.4 |
| 155 | 5 | 37.9 |
| | | |

The access time in Table 3 can be estimated with this model equation:

$$T_{av} = a + bN_r + cN_c + dN_cN_r$$

where

Nr = number of matrix rows    Nc = number of matrix columns

a = 15.7766 E-3    c = -754.840 E-9
b = 33.9161 E-8    d = 21.2832 E-6

This model equation has a maximum relative error of about 6.7%, but the RMS error is about 1.5%, so it may be useful to estimate access times for matrices not shown in the table. The graph below shows the points used to generate the model equation, which are the points from table 3.

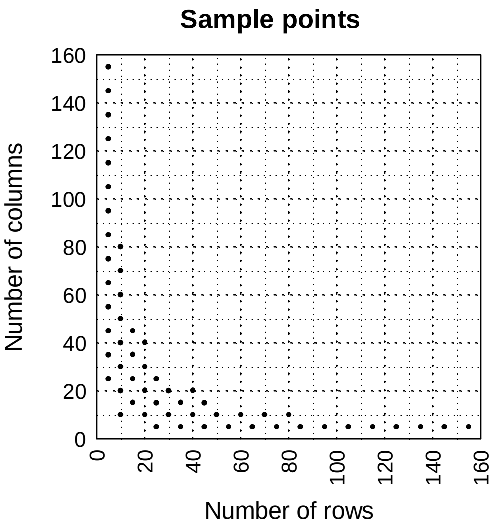


Table 4 shows the mean time to access an element in a few of the matrices from table 3, but using indirection instead of accessing the elements directly. The indices are local variables. The mean overhead, compared to direct access, is about 5.7 mS.

***Table 4***
***Mean matrix element access time, indirect access***

| Matrix Row | Matrix Columns | Mean acces time, single element (mS) | Indirect access overhead (mS) |
|---|---|---|---|
| 5 | 25 | 23.8 | 5.0 |
| 5 | 105 | 32.6 | 5.6 |
| 5 | 145 | 36.9 | 5.6 |
| 10 | 40 | 30.4 | 5.5 |
| 10 | 70 | 36.4 | 5.3 |
| 15 | 15 | 27.2 | 6.6 |
| 25 | 5 | 24.9 | 5.8 |
| 25 | 25 | 35.2 | 5.8 |
| 40 | 10 | 31.3 | 6.2 |
| 70 | 10 | 38.7 | 5.8 |
| 105 | 5 | 36.3 | 5.7 |
| 145 | 5 | 42.1 | 5.7 |

### [7.44] Local subroutines execute faster than global subroutines

TI Basic subroutines can be defined outside of the calling routine (an global subprogram), or they can be defined within the calling routine (a local subprogram). As an example, this program *t()* calls the external function *sub1()*:

```
t()
Prgm
Local k,r
For k,1,500
 sub1(k)→r
EndFor
EndPrgm

sub1(x)
Func
Return x^2
EndFunc
```

*sub1()* would be defined locally in *t()* like this:

```
t()
Prgm
Local k,r,sub1

Define sub1(x)=Func
 Return x^2
EndFunc

For k,1,500
 sub1(k)→r
EndFor
EndPrgm
```

Local subroutine calls are slightly faster, as this table shows.

| | Global | Local |
|---|---|---|

| | | |
|---|---|---|
| Unarchived | 37.58 mS | 37.04 mS |
| Archived | 38.02 mS | 36.94 mS |

The times in the table are for one function call, averaged over 500 calls as shown in the examples above. The test routines ran on a HW2 TI-92 Plus, AMS 2.05. The local routine, archived, is about 3% faster than the global routine. This is not significant for most applications, but if your program makes thousands of calls to the subroutine, you may save a few seconds.

Defining the subroutine locally reduces the number of variables in a folder, and makes it a bit easier to remove the program if needed. If the subprogram has no use outside of its calling routine, there is little point to defining it globally.

### [7.45] Display text in multiple fonts, and Pretty-Print expressions

TI Basic programs are usually limited to displaying text in a single font size. However, E.W. ( author also of the EQW equation writer) has written a function to display text in 3 font sizes, as well as show expressions in pretty-print on any display screen. Much of the following description is from E.W.'s header in the C source code. You can get this function, called *write()*, from Bhuvanesh Bhatt's web site at http://tiger.towson.edu/~bbhatt1/ti/. The C source code is included.

The zip file includes versions for both the TI-89 (*write.89z*) and the TI-92 Plus (*write.9xz*). Make sure you use the correct version for your calculator. The examples in this tip show screen shots from the TI-92 Plus; screen shots from the TI-89 are similar.

*write()* supports these features:

- Write a line of text in one of three fonts and four modes.
- Write multiple text lines with the different fonts and modes.
- Write a line of text mixed with pretty-printed expressions.
- Write multiple lines of text mixed with pretty-printed expressions.

*write()* displays the arguments on the current active screen. It will only write inside the screen client area, that is, it will not write in the toolbar display or the status line area. The active screen may be the home screen, the Program I/O screen, or the Graph screen. Your program is responsible for displaying the screen before calling *write()*. These commands are used to display the different screens:

| | |
|---|---|
| Home screen: | *DispHome* |
| Program I/O screen: | *Disp* |
| Graph screen: | *DispG* |

There are three possible syntaxes for *write()*:

Syntax 1:

```
write(x, y, string [,font_size] [,mode])
```

Syntax 2:

```
write(x, y, {string, string, ...} [,font_size] [,mode])
```

Syntax 3:

```
write(x, y, {format_string, "expr",...,format_string,"expr"} [,font_size] [,mode])
```

The first syntax writes a single string. The second syntax writes the strings as multiple lines. The third syntax writes expressions, or lines of text mixed with expressions. The square brackets indicate that the arguments *font_size* and *mode* are optional and need not be included. However, if you include the *mode* argument, you must also specify the *font_size* argument.

*x* and *y* specify the starting position of the text in pixel coordinates. x is the column coordinate, and y is the row coordinate. The upper left coordinate of the screen has coordinates x = 0 and y = 0. Negative values of *x* and *y* cause the string to be displayed outside of the visible screen area.

*string* specifies the string to be displayed.

*font_size* specifies the size of the displayed text:

0 = small
1 = medium (the 'normal' font size; default)
2 = large

*mode* specifies one of five settings which control the appearance of the displayed text:

0 = white text on black background (over-write mode)
1 = black text on white background (OR mode; default)
2 = black text on white background (XOR mode)
3 = gray text on white background
4 = black text on white background (over-write mode)

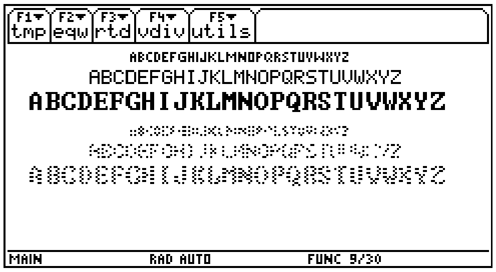
In over-write mode, the text and background replace any pixels previously displayed . In OR mode, the text and background pixels are logically OR'd with existing pixels, which means that the pixel at a given location will be 'on' (black) if the previous pixel *or* the text pixel is on. In XOR mode, the resulting pixel is off only if the original background pixel and the text pixel are the same. The over-write mode could be considered an 'opaque' mode, while the OR mode is a 'transparent' mode.

The third syntax and its arguments are described in an example below, after some examples for the first two syntaxes.

### *Syntax 1 example: Single lines, different font sizes, black and gray text*

This example uses the first syntax to display some sample text in three font sizes, with modes 1 and 3.

```
clrio                         © Clear Program I/O screen

"ABCDEFGHIJKLMNOPQRSTUVWXYZ"→s

disp                          © Display Program I/O screen
util\write(6Ø,3,s,Ø,1)        © Write lines in mode 1 (black on white)
util\write(4Ø,12,s,1,1)
util\write(1Ø,23,s,2,1)

util\write(6Ø,4Ø,s,Ø,3)       © Write lines in mode 3 (gray)
util\write(4Ø,49,s,1,3)
util\write(1Ø,6Ø,s,2,3)
```

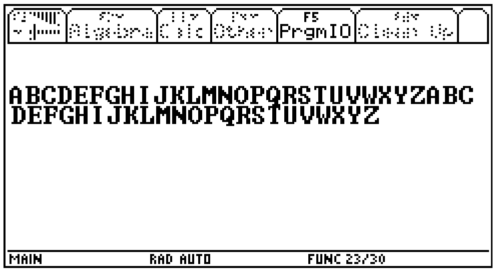


```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
ABCDEFGHIJKLMNOPQRSTUVWXYZ
ABCDEFGHIJKLMNOPQRSTUVWXYZ
ABCDEFGHIJKLMNOPQRSTUVWXYZ
ABCDEFGHIJKLMNOPQRSTUVWXYZ
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

For these examples, the *write()* function is stored in the *util* folder, so the function calls use that folder specification.

If the length of the text line exceeds the display width, the text wraps around to the next line, as shown with this example:

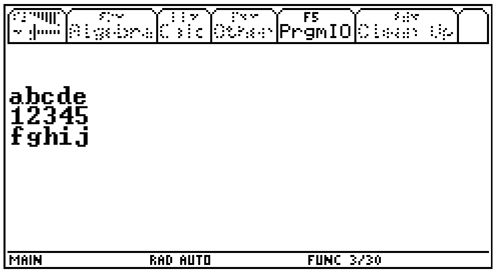
```
clrio
disp
"ABCDEFGHIJKLMNOPQRSTUVWXYZ"→s
util\write(Ø,2Ø,s&s,2,1)
```



```
ABCDEFGHIJKLMNOPQRSTUVWXYZABC
DEFGHIJKLMNOPQRSTUVWXYZ
```

You may also embed the carriage return character, char(13), in strings. This will force a line break, as shown in this example.

```
clrio
disp
char(13)→cr
"abcde"&cr&"12345"&cr&"fghij"→s
util\write(Ø,2Ø,s,2,1)
```

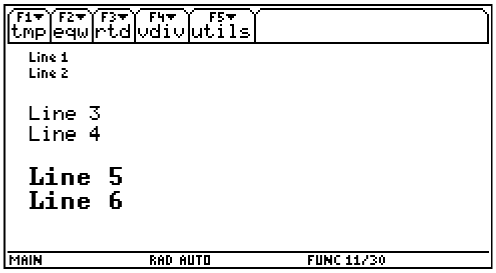


```
abcde
12345
fghij
```

***Syntax 2 example: Multiple lines***

The next example uses the second syntax to display multiple lines with one call to *write()*,

```
clrio
disp
util\write(10,3,{"Line 1","Line 2"},0,1)
util\write(10,30,{"Line 3","Line 4"},1,1)
util\write(10,60,{"Line 5","Line 6"},2,1)
```



```
Line 1
Line 2

Line 3
Line 4

Line 5
Line 6
```

### *Syntax 3 example: Pretty-print expressions*

As mentioned above, this syntax is used to display expressions:

```
write(x, y, {format_string, "expr",...,format_string,"expr"} [,font_size] [,mode])
```

*format_string* is string which contains one or more control characters. The control character char(174) will be replaced by the next expression *expr* in the list. char(174) is the circle-R character "®". This syntax allows the display of multiple expressions on one display line, and mixing text strings and expressions on the same line. In its simplest form, this syntax displays a single expression:

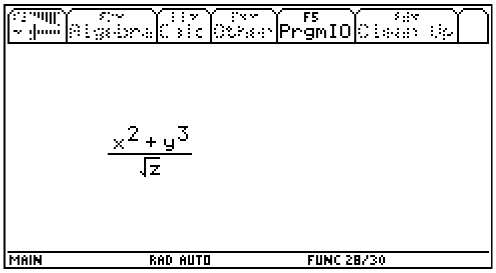
```
char(174)→c
util\write(50,40,{c,"(x^2+y^3)/√(z)"})
```

The control code is saved as the variable *c* for convenience, but this is not necessary. If you are coding directly on the calculator, you can insert the "®" character with [CHAR], [3], [N]. If you are coding in GraphLink, you cannot create the "®" character, and must instead use char(174).

*expr* is any valid expression, which is passed as a string.

*font_size* is ignored with this syntax. Expressions are always displayed with font size 1, the standard font. The *mode* argument is used, however. In mode 0 (white text on black background), only the text characters are displayed, and graphic elements such as divisor bars and root symbols may disappear. In mode 3 (gray characters), the expression may be completely illegible.

This code segment results in the following display:

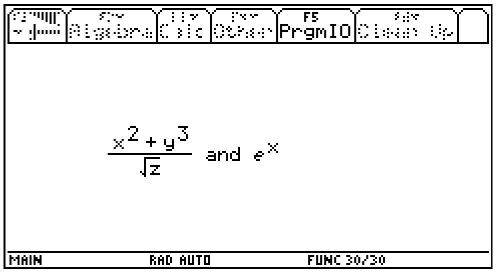


$$\frac{x^2+y^3}{\sqrt{z}}$$

Note that the expression is passed as a string, and the default values for *font_size* and *mode* are used.

Up to ten control characters may be used in a single *format_string*. This example shows a *format_string* which contains both control characters and text:

```
char(174)→c
util\write(5Ø,4Ø,{c&" and "&c,"(x^2+y^3)/√(z)","e^(x)"})
```

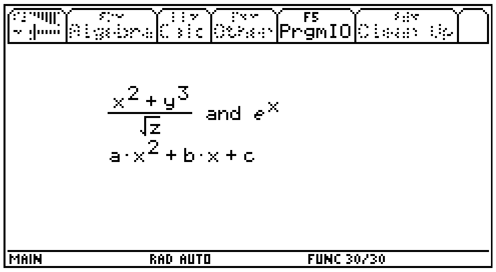


$\frac{x^2+y^3}{\sqrt{z}}$ and $e^x$

In this case *format_string* is built by concatenating the control characters with the text using the "&" operator, with the expression *c&" and "&c*, which results in the string "® and ®". Since there are two control characters, the list must include two expressions. In general, the number of expressions must equal the number of control characters.

This syntax for *write()* can also be used to display pretty-print expressions on multiple lines. Each *format_string* starts a new display line, as this example shows:

```
char(174)→c
util\write(5Ø,2Ø,{c&" and "&c,"(x^2+y^3)/√(z)","e^(x)",c,"a*x^2+b*x+c"})
```



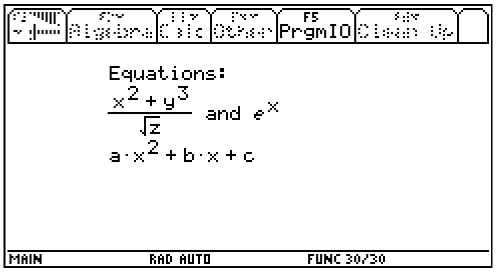
$\frac{x^2+y^3}{\sqrt{z}}$ and $e^x$

$a \cdot x^2 + b \cdot x + c$

Note that the second *format_string*, which is just *c*, causes the third expression to be displayed on a new line.

Finally, this syntax can also be used to display text on some lines, and expressions on others. This example shows the text "Equations:" on the line before the expressions:

```
char(174)→c
util\write(5Ø,1Ø,{"Equations:",c&" and "&c,"(x^2+y^3)/√(z)","e^(x)",c,"a*x^2+b*x+c"})
```



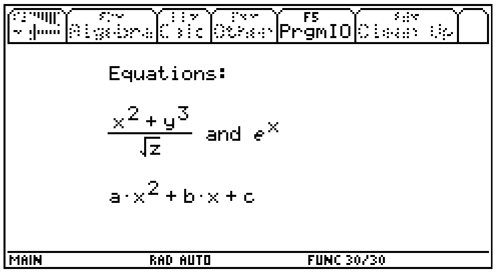
Equations:

$\frac{x^2+y^3}{\sqrt{z}}$ and $e^x$

$a \cdot x^2 + b \cdot x + c$

You can use empty strings ("") to insert blank lines:

```
char(174)→c
util\write(5Ø,1Ø,{"Equations:","",c&" and "&c,"(x^2+y^3)/√(z)","ℯ^(x)","",c,"a*x^2+b*x+c"})
```



Equations:

$\frac{x^2+y^3}{\sqrt{z}}$ and $e^x$

$a \cdot x^2 + b \cdot x + c$

### *C source code*

```
// C Source File
// Created 5/22/2001; 10:07:44 PM
// Author: E.W.


#define OPTIMIZE_ROM_CALLS    // Use ROM Call Optimization

//#define SAVE_SCREEN           // Save/Restore LCD Contents

//#include <tigcclib.h>         // Include All Header Files

//short _ti89;                  // Produce .89Z File
//short _ti92plus;              // Produce .9XZ File


#include <nostub.h>
#include <estack.h>
#include <wingraph.h>
#include <error.h>
#include <args.h>
#include <string.h>
#include <graph.h>

//#include <tigcclib.h>

#define WMAIN ((WINDOW*)(*((long*)(*((long *)(*((long*)0xC8)))))));

int _ti89,_ti92plus;
short GetInt(unsigned char **exp)
{
short value,err;
if((err=estack_to_short(*exp, &value))!=1) {
if(err==0)
ER_throwVar(830);
ER_throwVar(40);
}
```

```
*exp=next_expression_index(*exp);
return value;
}
void _main(){
  unsigned char *ap,tag,*ap1,*ap2,*ap3,*expr[10];
  short
f=1,c=-1,x,x1,attr,y,y1,NumArguments,bot,top,width,sindex,index,wi[10],tmax,bmax,i;
  char *s,*si[11],*s1;
  int error;
  ERROR_FRAME err_frame;
  WINDOW *w=WMAIN;

//  clrscr();

  InitArgPtr(ap1);
  NumArguments=remaining_element_count(ap1);
  if(NumArguments<3)
    ER_throwVar (930);
  if(NumArguments>5)
   ER_throwVar (940);
  x=GetInt(&ap1);
  y=GetInt(&ap1);
  ap=ap1;
  ap1=next_expression_index(ap1);
  if(NumArguments>3)
   f=GetInt(&ap1);
  if(NumArguments>4)
   c=GetInt(&ap1);
  if(f>2||c>4)
   ER_throwVar (40);
  if(f<0) f=1;
  tag=*ap;
  ap1=top_estack;
  if(tag==LIST_TAG) ap--;
  w->Flags|=WF_TTY ;
  if (!(error=ER_catch(err_frame))) {
if(c>=0)
attr=WinAttr(w, c);
if(f!=1)
WinFont (w, f);
while(*ap!=END_TAG) {
if(*ap!=STR_TAG) ER_throwVar (130);
s=*si=GetStrnArg(ap);
index=0;
sindex=1;
tmax=f+2;
bmax=f+4;
x1=x;
while((s=strchr(s,174))!=NULL&&index<=10) {
if(tag!=LIST_TAG) ER_throwVar (90);
if(*ap==END_TAG) ER_throwVar (930);
if(*ap!=STR_TAG) ER_throwVar (130);
*s++=0;
si[sindex++]=s;
if(*ap==STR_TAG) {
push_parse_text(GetStrnArg(ap));
}
Parms2D(expr[index]=Parse2DExpr (top_estack, 0), &wi[index],&bot,&top);
top--;
bot++;
if(bot>bmax) bmax=bot;
if(top>tmax) tmax=top;
index++;
}
if(y>w->Client.xy.y1-w->Client.xy.y0) break;
y=y+tmax+1;
y1=y+bmax+1;
if(y1<0) continue;
for(i=0;i<sindex;i++) {
WinStrXY(w,x1, y-f-2,si[i]);
x1=x1+DrawStrWidth(si[i], f);
if(index--) {
Print2DExpr(expr[i],w,x1,y);
x1+=wi[i];
if(f!=1)
WinFont (w, f);
}
}
if(tag!=LIST_TAG) break;
top_estack=ap1;
y=y1;
}
ER_success();
  }
  w->Flags&=~WF_TTY ;
  if(f!=1)
   WinFont (w, 1);
  if(c>=0)
   WinAttr (w, attr);
  if(error) ER_throwVar (error);

//  ngetchx();
}
```

### [7.46]  'Custom' command makes menu labels for trapped function keys

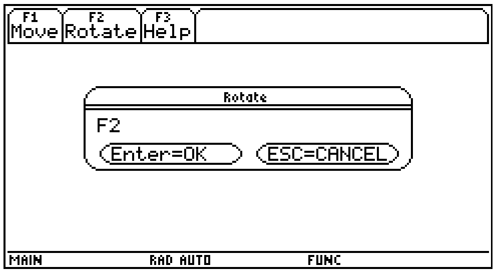
This tip shows how to use the *Custom* command to create labels for function keys used in interactive programs. An example might be a program which uses the Graph screen to display information, and the user can press one of several keys to perform program features. For example, suppose you have written a program which manipulates graphic objects on the screen. The objects can be moved and rotated, and there is a help feature. You want to provide quick, obvious access to these features.  This program shows the basic idea.

```
tooltab(m)
Prgm
© Custom toolbar with key trap demonstration
© (m) m is program name to restore menu, as a string
© 13jan02/dburkett@infinet.com

local key             © Pressed key code

dispg                 © Display graph screen

custom                © Display toolbar tabs
 title "Move"
 title "Rotate"
 title "Help"
endcustm
custmon               © Display toolbar

© Loop to wait for key presses
loop
 getkey()→key         © Get pressed key code
 if key=264:exit      © Exit when [ESC] pressed

 if key=268 then      © [F1] key
  dialog
   title "Move"
   text "F1"
  enddlog

 elseif key=269 then  © [F2] key
  dialog
   title "Rotate"
   text "F2"
  enddlog

 elseif key=27Ø then   © [F3] key
  dialog
   title "Help"
   text "F3"
  enddlog

 endif

endloop

if dim(m)≠Ø:expr(m)    © Restore menu

disphome               © Display home screen

EndPrgm
```

The basic principles are:

1. Display the graph screen. You may need to clear the axes and make other settings, depending on your program.
2. Display a custom toolbar with only *Title* entries, no *Item* entries.
3. Execute a loop which includes a *getkey()* call get key press codes, and test the key code for function keys. The loops traps the *function* key presses.

When *tooltab()* is executed and [F2] is pressed, this screen is shown:



The dialog boxes are only examples; your program would instead perform the code needed to perform the desired function.

Another feature of *tooltab()* is restoring a custom menu. *tooltab()* replaces any custom menu with its own, so it would be considerate to restore the user's custom menu. No TI Basic program which changes the screen from Home to some other can restore a custom menu, but we can at least make it convenient to do by pressing [CUSTOM] after the program ends. *tooltab()* accomplishes this by accepting a program name as a string argument *m*, then executing

```
if dim(m)≠Ø:expr(m)    © Restore menu
```

before exiting. For example, suppose I have a program to set up a custom menu called *main\custom92()*, then I would use this call:

```
tooltab("main\custom92()")
```

After exiting *tooltab()*, I just press [CUSTOM], and my menu is restored. Note that I use the folder specifier *main*, so *custom92()* is executed regardless of which folder from which *tooltab()* is executed. If you would rather not use this feature, just call *tooltab()* with an empty string:

```
tooltab("")
```

### [7.47] Global variables not evaluated when local variable has the same name

Unexpected behavior can result when global variable names are used as input in a *Request* statement. If the program has defined a local variable of the same name, the global variable will not be evaluated. This situation arises, for example, when you use global variables to save floating-point numbers. The program below demonstrates:

```
t()
Prgm

local x

clrio

dialog
 request "x",x
enddlog

expr(x)→x
disp "x is",x

EndPrgm
```

Expected operation ensues if you supply a number for *x* at the *Request* prompt. For example, if you enter 4.22 at the prompt, then the Program I/O screen shows

```
x is
4.22
```

However, suppose you have previously stored the value 4.17723887656 to a global variable *x*, for use in this program. To avoid re-typing the value, you just enter x at the prompt. In this case, *x* is not evaluated, and the Program I/O screen shows

```
x is
x
```

A work-around is to use unusual local variable names, in the optimistic hope that the program's user will not have a global variable of the same name. Characters from the International sub-menu of the CHAR menu may be appropriate.

### [7.48] Create functions in programs

Creating functions in a program is straightforward, as long as you really define them as functions and not expressions. For example, suppose we have created this function

```
chi(a,x,b)
Func
when(x>a and x<b,1,Ø)
EndFunc
```

and we want to use it to create a function g(x) of the form

```
chi(a,x,b)*f(x)
```

which can be evaluated within the program, or at the entry line. We define the function within the program like this:

```
chi(1,x,2)*x^2→g(x)
```

*this works, as a function*

and not like this:

```
chi(1,x,2)*x^2→g
```

*this DOES NOT work, as an expression*

The second example results in an *Undefined variable* message if we try to evaluate *g* at the entry line.

If you want g(x) to be a local function, you must declare *g* with the *Local* command.

### [7.49] Read and write text variables

There are no built-in commands or functions to read or write the contents of text variable, but Bhuvanesh Bhatt has written two flexible C routines to fix that: *TxtWrite()* and *TxtRead()*. Get them, along with documentation, at *http://tiger.towson.edu/~bbhatt1/ti/*.

From Bhuvanesh' documentation:

*Syntax:*

```
TxtWrite(expression_or_string,"varname","write_mode"[,str_format])
```

If you wish to prevent evaluation or auto-simplification of the expression, input it as a string. To insert a newline, just input a blank string.

The writing mode *write_mode* can be:
"w" or "write" if you want to create a new text variable or overwrite an existing one
"a" or "append" if you want to append to an existing text variable

*str_format* should be:
0 (default) if strings should be written without quotation marks
1 if quotation marks should be added.

```
TxtRead("varname")
```

*TxtRead()* returns the entire text variable as a list of strings. Do not try to read anything other than a Text variable. The maximum number of characters per line for *TxtRead()* is 512.

*Example:*

```
TxtWrite("Here is an expression","test","write")
TxtWrite(x^2+2,"test","append")
TxtWrite("Let's try a string with quotes","test","append",1)
```

Now the text variable *test* in the current folder will contain:

```
:Here is an expression
:x^2+2
:"Let's try a string with quotes"
:
```

*General comments:*

For the variable name, you can specify a variable in the current folder or you can give the full path. For *TxtWrite()*, if the folder you specify does not exist, you will be prompted to create it.

Any input errors (bad argument type, etc.) are shown in the status line.

*TxtRead()* is a C program, so you will need *IPR*, as well as *h220xtsr* if you have a HW2 calculator. See tip [12.2] for details.

### [7.50] Modify loop control variables within the loop

If you are familiar with other programming languages but not TI Basic, you may not be aware that you can modify all of the control variables of a *For ... EndFor* loop, within the loop block. While this is a dubious practice in terms of code readability and maintenance, it can sometimes result in smaller code.

For example, this loop will never terminate, because the index *i* never reaches the terminal value *n*:

```
1→n
For i,1,n
 i+1→n
EndFor
```

You can change the index step size within the loop:

```
1Ø→n
1→step
For i,1,n,step
 if i=4
  2→step
EndFor
```

For this example, *i* steps through 1, 2, 3, 4, 6, 8, 10.

The example below is a more realistic demonstration.  This segment of code processes a list of expressions *e* to return a list of the individual terms of the expression. Initially, list *e* has only one element, but it grows as terms are separated, and shrinks as they are removed to the output list *o*.

```
while dim(e)≠Ø

 for i,1,dim(e)                              © Process each element of 'e'
   e[i]→ex
   part(ex)→px
   augment(left(e,i-1),right(e,dim(e)-i))→e  © 'e' reduced by one element
   if px=Ø or px=1 then
    augment(o,{ex})→o
   elseif px=2 then
    part(ex,Ø)→pxØ
    if pxØ="+" then
     augment(e,{part(ex,1),part(ex,2)})→e    © 'e' increased by two elements here ...
    elseif pxØ="-" then
     augment(e,{part(ex,1),⁻part(ex,2)})→e   © ... and here
```

```
        else
         augment(o,{ex})→o
        endif
       endif
     endfor

    endwhile
```

The processing details are not relevant; the point is that the *For* loop will execute until the index reaches the current dimension of *e*, which is updated each pass through the loop.

# 8.0 String Variable Tips

## [8.1] Convert integers to strings without extra characters

When building strings to evaluate with *expr(),* you may need to insert integer strings in the string. For example, suppose you want to build the string "y*{n}*(x)", where *{n}* is an integer that is determined while the program is running. This won't always work:

```
"y"&string(n)&"(x)"
```

If the current mode is Exact/Approx mode is Exact, this works as expected. However, if the mode is Approx, and the user has set the Exponential Format mode to Engineering or Scientific, you'll get this:

`string(1)` results in "1.E0"

which causes problems if replaced in the example string above. Instead of using string(n), use

```
string(exact(n))
```

which will return just the integer, without decimal points, trailing zeroes, or the 'E' symbol, regardless of the mode settings.

## [8.2] String substitutions

Occasionally it is necessary to replace all occurrences of some substring within a string with a different string. For example, if we replace all of the "c"s in "acbc" with "x", we get "axbx". This code accomplishes this function:

```
strsub(s,so,sn)
func
©strsub(string,oldsub,newsub)
Local c,sl
""→sl
instring(s,so)→c
while c>Ø
 sl&mid(s,1,c-1)&sn→sl
 mid(s,c+dim(so))→s
 instring(s,so)→c
endwhile
return sl&s
endfunc
```

where *s* is the target string, *so* is the original pattern to be replaced, and *sn* is the replacement. So to use this program with my example, we would call it like this:

```
strsub("acbc","c","x").
```

*(credit declined)*

### [8.3] Creating strings that include quote characters

Suppose you want to store "abcde" in a string *str1*. This won't work:

```
""abcde"" → str1                won't work!
```

What actually gets saved in *str1* is this:

abcde∗""$^2$

which is interesting, because the 89/92+ is interpreting the input string as

```
abcde ∗ "" ∗ ""
```

However, these will work:

```
"""abcd"""->str1

string("abcde")->str1

char(34) & "abcde" & char(34) → str1
```

so that *str1* contains ""abcde"", as you would expect. 34 is the character code for the double quote symbol.

It is not straightforward to create strings that embed the double quote character, ", because that is also the character that delimits strings. Suppose you need to create the string

```
"ab"cd"
```

Just typing that in won't work: you'll get the *missing "* error message. This will work, though:

```
"ab""cd"            
```
*does work*

This will also work:

```
"ab" & char(34) & "cd"
```

because 34 is the character code for the double quote character.

You can include two or more consecutive quotes like this:

```
"ab""""cd"
```

In general, use two quote characters for each quote you want to create.

### [8.4] Count occurences of a character in a string

If you need to count how many times a particular character occurs in a string, use this:

```
charcnt(st,ch)
Func
©(string,char) count #char in string
Σ(when(mid(st,i,1)=ch,1,Ø),i,1,dim(st))
```

```
EndFunc
```

*st* is the string, and *ch* is the character to count. For example,

```
charcnt("abcdddef","d")            returns 3
```

*(Credit to Mike Grass)*

### [8.5] *string()* uses the current number display format

When *string()* converts a number to a string, it uses the current display format. This behavior is not explicitly described in the 89/92+ manual. While this is usually what you want to do when using *string()* to convert numbers for display, it is usually *not* what you want if you use string to convert a number for additional processing.

For example, suppose you want to extract the mantissa of of *x*, where x = 1.2345678901234E15. If the current display format is Fix 4, then string(x) returns "1.2346E15". If you use *left()* and *inString()* to extract the mantissa, you will get "1.2346", which is not the actual mantissa. You can retain most of the precision in *x* by first setting the format to Float 12, then string(x) returns "1.23456789012E15". However, note that you have lost the last two significant digits of ...34.

### [8.6] Convert strings to upper- and lower-case

The two functions below convert an alphanumeric string to all upper-case or all lower-case characters. This is useful, for example, to test strings entered by a user in a program.

Upper-case and lower-case character codes differ by 32, but converting the case is not as simple as adding or subtracting 32, because the string may also include characters which should not be changed. Only characters with lower-case equivalents are converted. In addition, the character set includes international characters, so the conversion routines handle those characters as well

The function to convert a string to lower-case:

```
casel(s)
Func
© (string) convert to lower case
© 7may02/dburkett@infinet.com

local ä,ï,ÿ

""→ÿ

for ä,1,dim(s)
 ord(mid(s,ä,1))→ï
 ÿ&char(when(ï≥65 and ï≤90 or ï≥192 and ï≤214 or ï≥216 and ï≤223,ï+32,ï))→ÿ
endfor

return ÿ

EndFunc
```

The function to convert a string to upper-case:

```
caseu(s)
Func
© (string) convert to upper case
© 7mayØ2/dburkett@infinet.com

local ä,ï,ÿ

""→ÿ

for ä,1,dim(s)
 ord(mid(s,ä,1))→ï
 ÿ&char(when(ï≥97 and ï≤122 or ï≥224 and ï≤246 or ï≥248 and ï≤256,ï-32,ï))→ÿ
endfor

return ÿ

EndFunc
```

| | | | |
|---|---|---|---|
| Some examples: | `caseu("abc")` | returns | `"ABC"` |
| | `casel("ABC")` | returns | `"abc"` |
| | `caseu("a+b=C")` | returns | `"A+B=C"` |

Both functions work by getting the code of each character in the string, with *ord()*, then adjusting the code by 32 only if the character has a lower-case equivalent, as determined by its code. Note that the *and* operator has higher priority than *or*, so no parentheses are needed.

The execution time in seconds can be estimated as 0.097c + 0.081, where *c* is the number of characters in the string. For example, a 10-character string converts in about 1.05 seconds.

### [8.7] Replacement for mid() and right() functions use position instead of count

The built-in *mid()* function has the syntax *mid(string, start [,count])* when used with a string argument. *count* characters are returned from *string*, beginning at character *start*. Sometimes it is more convenient to extract characters from a *start* position to an *end* position, instead of a number of characters. It is trivial to find

count = end - start + 1

but it may be useful to define it as a function:

```
mid1(s,n,m)
Func
©(s,n,m) return elements from n to m of s
©7novØ1/dburkett@infinet.com

mid(s,n,m-n+1)

EndFunc
```

For example, `mid1("abcdefghi",3,7)` returns "cdefg".

The built-in *mid()* function also operates on lists, and so does *mid1()*, since it calls *mid()*. For this reason, the default behavior of *mid1()* is the same as *mid()*.

We can also write a similar replacement function for *right()* which uses a position for the argument, instead of the number of elements:

```
right1(s,p)
Func
©(s,p) return right elements from p of s
```

```
©7novØ1/dburkett@infinet.com
right(s,dim(s)-p+1)
EndFunc
```

For example, `right1("123456789",7)` returns "789".

We don't need a replacement function for *left()*, because the number of elements used as the argument in that function *is* the position in the string.

## [8.8] Relational operators compare strings

The relational operators (=, <, >, etc.) work on strings as well as numbers, although this is not specified in the calculator *Guidebook*. For example

| | |
|---|---|
| "a" > "b" | returns *false* |
| "a" < "b" | returns *true* |
| "a" > "A" | returns *true* |
| "xyz" = "xyz" | returns *true* |

Comparisons for lists and matrices with string elements are performed element by element, for example

| | |
|---|---|
| {"a","b","c"} > {"c","b","a"} | returns {*false*, *false*, *true*} |

Strings are sorted in order of the TI-89 / TI-92 Plus character codes as found in Appendix B of the *Guidebook*. For example, these comparisons all return *true*:

| | |
|---|---|
| "A" < "a" | *character code 65 is less than code 97* |
| "<" < "a" | *character code 60 is less than code 97* |

If the strings are of unequal length, then shorter strings are 'less' than longer strings with identical initial characters; these expressions are true:

| | |
|---|---|
| "aa" < "aaa" | returns *true* |
| "a" < "azz" | returns *true* |

This behavior conforms to lexicographical (English dictionary) ordering, in which short words precede longer words. This character-code ordering may not necessarily be the ordering you want. As an example, the character codes for all the international characters are greater than the English alphabetic characters. This means that

| | |
|---|---|
| "cat" < "ä" | returns *true* |

The commands *SortA* and *SortD* also use these ordering rules when sorting lists and vectors with string elements.

# 9.0 User Interface Tips

## [9.1] Use icons in toolbars

Custom toolbar titles and labels are usually text. It is an undocumented feature of the 89/92+ that you can also use small graphics (icons) in place of the toolbar titles and labels. This code extract shows the method:

```
circ()
Prgm
ClrIO
Toolbar
Title circimg
Item radimg,r
Item areaimg,a
Item circimg,c
EndTBar

Lbl r
 ...

Lbl a
 ...

Lbl c
 ...

EndPrgm
```

In this example, the icons are the variables *circimg*, *radimg*, and *areaimg*. Note that the icon *circimg* is used in both the toolbar title, and in the third menu item. The icons are PIC variables, 16 pixels high by 16 pixels wide.

The icons can be created in a number of ways. You can use the pixel functions: *PxlCrcl*, *PxlLine*, *PxlOn* and *PxlText* in a program to create the image, then use *StoPic* to save the image. This program creates an icon, and saves it as *icon1*.

```
iconex()
Prgm
©Create & save a simple icon

©Clear the graph screen
clrgraph
clrdraw

©Draw a box
pxlline 1,1,1,16
pxlline 1,16,16,16
pxlline 16,16,16,1
pxlline 16,1,1,1

©Draw a circle in the box
pxlcrcl 8,8,6

©Draw an "E" in the circle
pxltext "E",5,5
```

```
©Save the icon
stopic icon1,1,1,16,16

EndPrgm
```

You can also use the *NewPic* command with a matrix to to build an icon.

Frank Westlake has written a program for the 92+ called *Image Editor* to create icons and other picture variables. You can get it here:

http://frank.westlake.org/ti/index.html

John Hanna's *Iview* program at

http://users.bergen.org/~tejohhan/iview.html

can be used to create icons, or to convert other PC graphics files to icons.

Note that you can use the Contents toolbar item in Var-Link to view PIC variables, including the icons you create.

*(credit to Mike Grass for toolbar example)*

### [9.2] User interface considerations

These suggestions will help make your programs easier to use, and less frustrating. The goal of any program is to enable the user to accomplish a task as quickly and efficiently as possible, and I think these ideas work towards that goal. In almost all cases these suggestions will make your program larger, but rarely make it slower. Not all suggestions are appropriate for all programs.

- Check user inputs for data type and range. If the input should be an integer, warn the user if he enters a string or floating point number, and give him the chance to try again. If the input should be between certain limits, and he enters a number outside those limits, display a warning and let the user try again.
- If a program needs lots of inputs from the user, save those as global variables so they can be used as defaults with Request in dialog boxes.
- Trap errors (with *Try ... EndTry*), and display useful warning messages. The message should display both *why* there is a problem, and *what* to do to fix it. Many errors can be trapped before the code even runs: think about what the program will do with various error conditions, or different values for variables.
- Use the Title statement in dialog boxes to help the user. For example, you can use titles of INPUT, RESULTS, ERROR and WARNING to make the message purpose more clear. I find that all caps is easier to read in the small font used in the dialog box title.
- Don't change the mode settings without notifying the user. The Mode settings include the angle mode (radians, degrees), display format, complex number format, graph type and so on. If you must change the modes, notice that *exact(), approx(),* $^r$, and ° can be used to temporarily over-ride the arithmetic mode and angle mode. If you have to change other modes, use *GetMode("ALL")* and *SetMode()* to save and restore the user's mode settings.
- For large, complex programs consider including a 'help' menu item or custom key. The help should explain how to use the program and any special features or limitations. The help should be built into the program so that the user can see it while running the program. Ideally, the help code and text

strings should be very modular, and your program should provide an option to delete the help system, since this uses up RAM very quickly. In this way, inexperienced users get the help they need to run your program, and they can remove it when they don't need it anymore.

- Set the application's folder within the program itself. This makes sure that all subprograms, functions and variables are available while the program is running. When the program exits, restore the folder so that the user doesn't have to, and isn't surprised to be in a different folder.
- Use local variables if at all possible. However, you will be forced to use global variables in some cases. Use DelVar to delete these automatically when the program exits. If you think that the user might want to keep them, give her the option to delete them when the program exits.
- If your program uses matrices or other data structures, it is most likely that those are not in the application's folder. So, when you prompt for the variable name, make sure that the user knows that he needs to enter the folder name, as well. Sometimes I prefer two separate Request statements, one for the folder name, and one for the variable name. Your program needs to be able to refer to variables by both the folder name and the variable name.
- If your program creates large data structures, check available memory with *getConfg()* before creating the variable. If there isn't much free RAM, warn the user.
- If your program uses large data structures, think about archiving them automatically, so they don't use so much of the calculator's RAM. If your program only needs to read from the data structure, it can remain permanently archived. However, if the program needs to write to the data structure, you will need to unarchive it, perform the write operation, then archive it again. If you try this approach, make sure that your program cannot execute a loop in which the variable is rapidly and repeatedly archived. In extreme cases this will 'wear out' the archive flash memory.
- Provide an 'exit' or 'quit' menu item in your program. This gives the program a chance to delete global variables, restore the mode settings, and restore the folder.
- If your program displays output on the program I/O screen, use *DispHome* when the program exits. This means the user doesn't have to press [green diamond] HOME to get back to the home screen.
- Consider providing a menu to let the user set the number display digits and exponential mode, for numeric results. Use *format()* to display the results based on the user's choice. In large programs with lots of output, I even let the user set different formats for different sets of variables.
- If possible, use functions and commands that are available on both the original 92 as well as the 89/92+. This means that more people can use your program. However, this is a common dilemma encountered whenever hardware and software improves. In some cases, this means that users with the newer systems pay a penalty in terms of execution speed or code size.
- Keep in mind that there are two broad types of users. The first type just wants an answer from your program from the command line, as quickly and easily as possible. This user needs dialog boxes to prompt for the inputs, and clearly labeled output. The second type of user might want to use your code in her own programs. This user wants your code as a function, not a program, so it can be easily called, and return its results to the user's program. One clean way around these conflicting requirements is to encapsulate your core functionality as a function, and supply an additional interface routine that provides prompts and labelled output.
- Provide documentation for your program. You might have written the greatest program in the history of coding, but if people can't figure out how it works, you might as well have saved yourself the effort. The documentation doesn't need to be a literary work of art.

In the end, it is your program, and you'll write it the way you want to. And writing a robust, efficient program is a lot of work. It is likely that if you use all of these suggestions, the amount of code needed to make the program a pleasure to use will be far greater than the code to actually do the work! Depending on the program, it might not be worth the effort. You just have to use your judgement.

### [9.3] Take advantage of *ok* system variable in dialog boxes

The *Dialog...EndDlog* structure always shows two button choices: [ENTER] and [ESC]. If [ENTER] is pressed, the system variable *ok* is set to 1. If [ESC] is pressed, *ok* is set to zero. You can use this information to exit the program if [ESC] is pressed, or return to a main program screen, or take some other appropriate default action.

### [9.4] Displaying more lines on the 92+ program I/O screen

The *Disp()* function is used to display strings on the program I/O screen. As more lines are displayed, previous lines scroll off the top of the display. Only 8 complete lines can be shown at once. This program can be used to legibly display 10 lines at once:

```
disprc(lcdrow,lcdcol,outstr)
Prgm
output 1Ø*lcdrow,6*lcdcol,outstr
EndPrgm
```

*lcdrow* is the row (0 to 9) at which to display the text.
*lcdcol* is the character column (0 to 39) at which to display the text.
*outstr* is the string to display

40 characters can be displayed on each line. Note that you can also use this method to quickly update display screens in your programs, when only a small part of the screen changes.

You can squeeze 11 lines on the display by changing `1Ø*lcdrow` to `9*lcdrow`, but this is not as legible because the lines are very closely spaced.

### [9.5] Default values for variables in *Request*

*Request* is used in dialog boxes for prompt for user input. You can save your users a lot of time if you provide default values, and remember those defaults. However, there is a problem if the variables are numbers, since *Request* returns strings. This code gets around this problem:

```
string(x) → x

dialog
 request "Enter x",x
enddlog

expr(x) → x
```

This code assumes that *x* has a numeric value on entry, which is the default. The *string()* function converts *x* to a string for use in *Request*. When the dialog box exits, *expr()* converts the string to a number so you can use it in calculations.

If *x* is a local variable, you need to initialize it to the default first. If *x* is a global variable, the last-used value is the default the next time you run your program. While this is very convenient, it does take up RAM.

If you do use a global variable, it needs to be initialized the first time you run the program, otherwise the dialog box will look like this:

```
Enter x: x
```

The variable name will be displayed. This can be avoided like this:

```
if GetType(x)="NONE":Ø->x
```

This statement is executed before the *string()* function. The variable is initialized if it doesn't exist, otherwise it is unchanged.

If your program has a lot of inputs, consider saving them as a list. This creates a little more work for you, as the programmer, but it reduces the clutter of global variables in a folder. As an example, suppose that your program needs four inputs from the user. This example shows the basic idea.

```
local w,x,y,z                      © Define local variables for user inputs

if gettype(defaults)="NONE"        © If the user defaults variable doesn't exist,
 {1,2,3,4}→defaults                © ... create and initialize it

string(defaults[1])→w              © Initialize local copies of variables
string(defaults[2])→x
string(defaults[3])→y
string(defaults[4])→z

Dialog                             © Prompt for new values
 Request "Enter w",w
 Request "Enter x",x
 Request "Enter y",y
 Request "Enter z",z
EndDlog

if ok=Ø:return                     © Just quit if user presses [ESC]

expr(w)→w : w→defaults[1]          © Convert string to expression; update global copy
expr(x)→x : x→defaults[2]
expr(y)→y : y→defaults[3]
expr(z)→z : z→defaults[4]
```

In this example, *defaults* is the name of the global variable which holds the default values. If your inputs are lists or matrices, see tip [3.24] for a method to store those data types in the default list.

## [9.6] Position cursor with char(2)

Use char(2) to position the cursor in strings. This idea can be used in text displayed on the command line, or in programs or functions. For example, if you define a custom menu item as

```
item "f("&char(2)&")"
```

then when the menu item is executed, the command line shows

```
f(|)
```

with the cursor between the parentheses, ready for you to enter your argument.

If you enclose the string with a pair of char(2) characters, then the string will be highlighted in the command line when you select the custom menu item. This program shows the idea:

```
menus()
Prgm

custom
 title "funcs"
 item "f("&char(2)&"xyz"&char(2)&")"
endcustm

custmon

EndPrgm
```

This creates a custom menu with one menu tab, labelled "funcs". The menu has a single item, f(xyz). When you select this menu item, the argument *xyz* is highlighted. You can press [ENTER] to execute f(xyz), or you can press [CLEAR] or [BACKSPACE] to clear *xyz* and enter a different argument.

A variation on this theme is to enclose "ans(1)" between the pair of char(2) characters. When the menu item is chosen, *ans(1)* is highlighted, so that you can accept it by pressing [ENTER], or erase it with one keystroke ([BACKSPACE] or [CLEAR]). This example shows *expand()* used with this method:

```
Item "expand("&char(2)&"ans(1)"&char(2)&")"
```

This technique saves keystrokes if you frequently use *expand()* or other CAS functions on previous results in the history display.

There is another method to create the char(2). This method must be used on the calculator, and programs which include char(2) created in this way cannot be sent to a PC with GraphLink, because GraphLink does not convert the character correctly.

1. Enter this in the command line and press [ENTER]: char(2)&char(2)&char(2)&char(2)&char(2)
2. Highlight the first answer and press [ENTER]. A string with three characters is highlighted. Copy this string to the clipboard.
3. In the program editor, paste the clipboard contents into a string (between double quotes). Only one highlighted instance of the character is pasted. Press [RIGHT] and continue typing in the program.

*(Credit to Glenn Fisher, submitted by Larry Fasnacht; highlight method by Bhuvenesh Bhatt, ans(1) variation by Bez, char(2) creation method by Kevin Kofler)*

### [9.7] Creating 'dynamic' dialog boxes

You may run into a programming situation in which you want to prompt for user input with the *Request* function, but you don't know the variable in advance. You can use *expr()* with a string argument to create the dialog box, like this:

```
expr("request " & char(34) & promptn & char(34) & "," & vname)
```

where *promptn* is the prompt string, and *vname* is the variable name as a string. For example, if

promptn = "what?"
vname = "myvar"

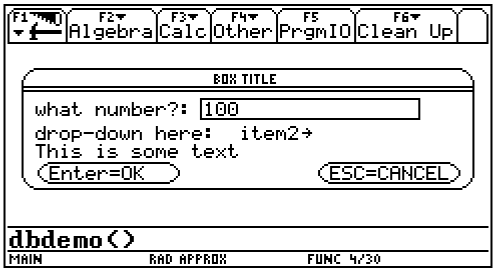
then the expression results in a dialog box like this:

```
what?:
```

and the string that the user enters will be stored in *myvar*. The value of this approach is that the actual prompt string and variable name can be stored in *promptn* and *vname* by preceding code, then the single *expr()* executes the dialog box. In other words, you can change the contents of a dialog box as needed, depending on run-time conditions.

This idea can be extended to all functions that can be used in a dialog box: *Text*, *DropDown*, and *Title*, in addition to *Request*. The basic principle is to build the complete *Dialog ... EndDlog* block up as a string. Each program line is separated by ":".

This dialog box demonstrates the use of all four dialog box functions.



BOX TITLE
what number?: 100
drop-down here:  item2→
This is some text

dbdemo()

To explain each option, as well as make it easier to build the dialog box, I have defined these functions:

| | |
|---|---|
| *dbttl():* | Dialog box title (Title) |
| *dbreq():* | Dialog box request (Request) |
| *dbdrd():* | Dialog box drop-down menu (DropDown) |
| *dbtxt():* | Dialog box text string (Text) |
| *dbend():* | Terminate the dialog box string |

These are the steps you use in your program to create the dialog box:

1. Initialize a string variable to start building the dialog box string
2. Call the four functions above, as needed, to make the dialog box you want
3. Terminate the dialog box string
4. Display the dialog box with *expr()*

The code shown below creates my dialog box example.

```
dbdemo()
Prgm

©dbdemo() - dynamic dialog box demo
©31 aug 99/dab
©dburkett@infinet.com

©Define local variables
local promptn,vname,boxtitle,sometext,ddtitle,dditems,ddvar,dbox

©Initialize dialog box items
"what number?"→promptn
"myvar"→vname
"BOX TITLE"→boxtitle
```

```
"This is some text"→sometext

"drop-down here:"→ddtitle
{"item1","item2"}→dditems
"dropvar"→ddvar

©Initialize the dialog box string
"dialog"→dbox

©Build the dialog box string
dbttl(dbox,boxtitle)→dbox
dbreq(dbox,promptn,vname)→dbox
dbdrd(dbox,ddtitle,dditems,ddvar)→dbox
dbtxt(dbox,sometext)→dbox

©Terminate the dialog box string
dbend(dbox)→dbox

©Display the dialog box
expr(dbox)

EndPrgm
```

In this example, I use the local variable *dbox* to hold the dialog box string. Note that the *dbox* is initialized to "dialog"; you must always initialize your dialog box string just like this.

After *dbox* is initialized, I call each of the four functions to create the title, a request, a drop-down menu and some text. Note that the first argument of each function is the dialog box string *dbox*. Each function simply appends the appropriate string to the current *dbox*. The table below shows the arguments for each function.

| **Description** | **Call convention** | **Arguments** |
|---|---|---|
| Create box title | dbttl(db,titletext) | db: dialog box string<br>titletext: string to use for title |
| Add Request | dbreq(db,promptn,vname) | db: dialog box string<br>promptn: prompt string<br>vname: variable name as a string |
| Add drop-down menu | dbdrd(db,prmpt,ddlist,ddvar) | db: dialog box string<br>prmpt: prompt string<br>ddlist: list of menu item strings<br>ddvar: variable name as a string |
| Add text | dbtxt(db,txt) | db: dialog box string<br>txt: text string |

Here is the code for the functions:

```
dbttl(db,titletxt)
func
db&":title "&char(34)&titletxt&char(34)
Endfunc

dbreq(db,promptn,vname)
func
db&":request "&char(34)&promptn&char(34)&","&vname
Endfunc
```

```
dbdrd(db,prmpt,ddlist,ddvar)
func
db&":dropdown "&char(34)&prmpt&char(34)&","&string(ddlist)&","&ddvar
Endfunc

dbtxt(db,txt)
func
db&":text "&char(34)&txt&char(34)
Endfunc

dbend(db)
func
db&":enddlog"
Endfunc
```

Note that you don't really need to use these functions, instead, you can just build the dialog box as a big string and use it as the argument to *expr().* This is a better approach if you only have a single dialog box.

## [9.8] Dialog box limitations

Dialog boxes can include *Text*, *DropDown* and *Request* functions. There are limits to the number of functions you can include in a single Dialog box:

| | |
|---|---|
| Text: | 10 maximum |
| DropDown: | 9 maximum |
| Request: | 7 maximum |
| Title: | 1 maximum |

This table shows the maximum number of *Request* functions for combinations of *Text* and *DropDown* functions. For example, if you have four *Text* functions and three *DropDown* functions, you can have as many as two *Request* functions.

| Number of Text lines ⇨ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Number of DropDown lines⇩ | | | | | | | | | | | |
| 0 | 7 | 6 | 6 | 5 | 4 | 3 | 3 | 2 | 1 | 1 | 0 |
| 1 | 6 | 6 | 5 | 4 | 3 | 3 | 2 | 1 | 0 | 0 | |
| 2 | 5 | 5 | 4 | 3 | 2 | 2 | 1 | 0 | | | |
| 3 | 5 | 4 | 3 | 2 | 2 | 1 | 0 | | | | |
| 4 | 4 | 3 | 2 | 2 | 1 | 0 | | | | | |
| 5 | 3 | 2 | 1 | 1 | 0 | | | | | | |
| 6 | 2 | 1 | 0 | 0 | | | | | | | |
| 7 | 1 | 0 | | | | | | | | | |
| 8 | 0 | | | | | | | | | | |
| 9 | 0 | | | | | | | | | | |

Here are some other Dialog box property limitations:

| Function | Properties |
|---|---|
| Title | • Maximum length of Title string is 50 characters<br>• The Title string may be empty; Title "" |
| Text | • Maximum length of Text string is 30 characters.<br>• If a Dialog box contains Text functions, at least one must contain text. |
| DropDown STR,LIST,VAR | • STR may be empty.<br>• The maximum length of STR depends on the length of the widest element in LIST. If STR has 36 characters, LIST elements must have zero characters. If STR has 0 characters, LIST elements may have up to 30 characters. Otherwise, the sum of the number of characters of STR and the widest element of list must be less than or equal to 35.<br>• LIST may not have any elements wider than 30 characters.<br>• LIST may have a single empty string.<br>• LIST may not have more than 2000 elements. |
| Request STR,VAR | • STR may have zero characters<br>• STR can be up to 20 characters, maximum. |

*(Credit to Frank Westlake)*

## [9.9] Display all 14 significant digits

The TI89/92+ perform floating point calculations with 14 significant digits of resolution. However, you cannot set the display mode format to show all these digits. The regression calculations, however, do display all 14 digits.

You can see all 14 digits for a number in the history area, though - just return it to the command line. For example, if the display mode is set to Float 12, then entering 'pi' will display

```
3.14159265359
```

but if you select it in the history area, it is shown in the command line as

```
3.1415926535898
```

Note that, in general, these 2 extra digits are probably not accurate or significant. Any computer or calculator that performs floating point arithmetic includes one or two 'guard digits'. These digits are used during calculations to eliminate or reduce round-off errors, especially during long calculation sequences. That is why the 89/92+ use them during calculations, but do not normally let you display them.

## [9.10] Group fraction digits for easier reading

If you often work with numbers with many digits, it is tedious and tiresome to read the numbers and copy them without making a mistake. These two routines help somewhat, by converting the number to a string with spaces between the fractional digits. I show two routines:

grpfrc(x,f) Is a function that returns *x* as a string, with groups of *f* digits. This routine is best used in a program, for displaying results.

grpfrcan() Shows the latest answer in the history display as a string with grouped fraction digits. This routine is best used from the command line, to display the last answer. The number of group fraction digits is fixed, but can be easily changed in the code.

Some examples:

| | | |
|---|---|---|
| grpfrc(1.23456789012,3) | returns | "1.234 567 890 12 E0" |
| grpfrc(1.23456789012,4) | returns | "1.2345 6789 012 E0" |
| grpfrcan() | returns | "1.414 213 562 37 E0" |
| | when 1.41421356237 is the first item in the history display | |

My examples show the exponent 'E0' because I had the Exponent mode set to Engineering, but these routines work for any Exponent mode setting.

The code for *grpfrc():*

```
grpfrc(x,g)
func
©grpfrc() - Group fraction digits
©12 May 99/dab
©dburkett@infinet.com
©x: number to format
©g: number of fraction digits in group

local n,dp,el,ex,mn,fn,f1

©Convert argument to string
string(x)→n

©Initialize exponent string
""→ex

©Find decimal point and ᴇ, if any
instring(n,".")→dp
instring(n,"ᴇ")→El

©Get exponent or set end of string
if El≠Ø then
right(n,dim(n)-El+1)→ex
else
dim(n)+1→El
endif

©Get mantissa & fraction strings
left(n,dp)→mn
mid(n,dp+1,El-dp-1)→fn

©Separate fraction digits with space
""→f1
while dim(fn)>g
 f1&left(fn,g)&" "→f1
 right(fn,dim(fn)-g)→fn
endwhile

©Build & return final result
mn&f1&fn&" "&ex

Endfunc
```

And the code for *grpfrcan(),* which just calls *grpfrc()* with the latest history area result:

```
grpfrcan()
func
©Group fraction digits of ans(1)
©2 nov 99/dab
©dburkett@infinet.com
grpfrc(expr("ans(1)"),3)
Endfunc
```

Note that the first argument to *grpfrc()* in *grpfrcan()* must be *expr("ans(1)"),* not just *ans(1),* as you might expect. Otherwise, *grpfrcan()* just grabs the first history result the first time you run it, then rudely *modifies* your code with that answer, and always returns the same result.

Change the second argument, 3, to a different number of fraction digits if needed.

### [9.11] Access all custom menus from any custom menu

Custom menus are useful for frequently performed operations. Refer to the *TI89/92+ User's Guide*, page 303 for details on creating and using custom menus. You can create any number of custom menus, and display them by running the program that creates them.

If you have several custom menus for different topics, you can display all your custom menus from each custom menu. This makes it faster to switch between the menus. The basic idea is to make a menu title section, in each custom menu, which contains the names of all your custom menus. For consistency, this should be the last menu for all custom menus. Even better is to ensure that the tab for the custom menus is always the same key, for example [F8]. Suppose I have three custom menus, *cmenu1(), cmenu(),* and *cmenu3().* This is the code for each of the three custom menus:

```
cmenu1()
Prgm
Custom
title "Menu 1"
item "a":item "b":item "c":item "d"
title ""
title ""
title ""
title ""
title ""
title ""
title "Menus"
item "cmenu1()":item "cmenu2()":item "cmenu3()"
EndCustm
CustmOn
EndPrgm

cmenu2()
Prgm
Custom
title "Menu 2"
item "d":item "e":item "f":item "g"
title ""
title ""
title ""
title ""
title ""
title ""
title "Menus"
item "cmenu1()":item "cmenu2()":item "cmenu3()"
EndCustm
CustmOn
EndPrgm

cmenu3()
Prgm
Custom
title "Menu 3"
item "h":item "i":item "j":item "k"
title ""
title ""
title ""
title ""
title ""
title ""
title "Menus"
item "cmenu1()":item "cmenu2()":item "cmenu3()"
EndCustm
CustmOn
EndPrgm
```

These are very simple menus, just to show the idea. Each menu has one menu tab [F1] which displays the menu items. Each menu program also has the same title item for [F8], which displays all the menu names. Each menu program also includes enough `title ""` statements so that the Menus tab is always the [F8] key.

If menu 1 is currently displayed, then you can switch to menu 2 by pressing [F8], [2] then [ENTER].

*(Credit to Andrew Cacovean)*

### [9.12] Use one *Request* (in a Dialog box) for more than one variable

The *Request* command is used in Dialog boxes to get input from the user. However, there is a limit to the number of *Request* commands allowed in a Dialog box. In addition, data is naturally entered in pairs or triplets, such as point coordinates. By using one *Request* command to get two variables from the user, you reduce the number of *Request* commands, and the user can enter the data in a more natural way.

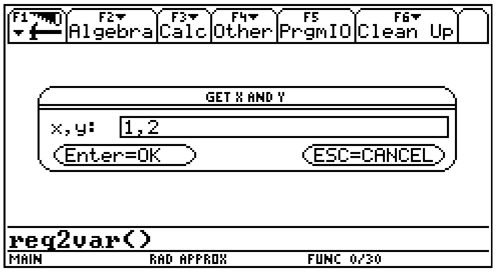
The two expressions entered by the user are separated by a comma. This program demonstrates the technique:

```
req2var()
Prgm
©Demo, get two variables in one Request
©14oct00 dburkett@infinet.com

local s,x,y,xy

©Set default values for x and y
1→x
2→y

©Combine x and y for default prompt
string(x)&","&string(y)→xy
```

```
©End up here if user omits comma between x and y
lbl tryagain

©Prompt for x and y
dialog
 title "GET X AND Y"
 request "x,y",xy                ©x and y will be in string xy
enddlog
if ok=Ø:return                   ©Let user quit by pressing [ESC]

©Find the comma position in xy
instring(xy,",")→s

©Test for comma
if s=Ø then
 dialog                          ©Display error message if no comma found
  title "ERROR"
  text "You must put a comma"
  text "between x and y"
 enddlog
 if ok=Ø:return                  ©Let user quit by pressing [ESC],
goto tryagain                    ©or try again by pressing [ENTER]
endif

©Convert x and y strings to expressions
expr(left(xy,s-1))→x
expr(right(xy,dim(xy)-s))→y

©Just for a demo, display x and y. Your application probably won't do this
dialog
 title "SHOW X AND Y"
 text "x is "&string(x)
 text "y is "&string(y)
enddlog

EndPrgm
```

Note that the copy of this program in the tlcode.zip file does not include all the comments.

When this program runs, this Dialog box is shown:



x,y: 1,2

req2var()

Note that the defaults of 1 and 2 are shown. The user presses [ENTER] to accept these defaults, or enters the new values, separated by a comma. Or, the user can press [ESC] to quit the program.

If the user forgets to enter the comma separating the two values, an error message dialog box is shown. The user can press [ESC] to quit the program at this point, or press [ENTER] to try again.

This idea can be extended to more than two variables. The program *req3var()* shows how to get three variables from one *Request* command.

```
req3var()
Prgm
©Demo, get three variables in one Request
©14octØØ dburkett@infinet.com

local s1,s2,x,y,z,xyz

©Set default values for x, y and z
1→x
2→y
3→z

©Combine x, y and z for default prompt
string(x)&","&string(y)&","&string(z)→xyz

©End up here if user omits commas between x, y and z
lbl tryagain

©Prompt for x,y and z
dialog
 title "GET X, Y AND Z"
 request "x,y,z",xyz                 ©x, y and z will be in string xyz
enddlog
if ok=Ø:return                       ©Let user quit by pressing [ESC]

©Find the comma positions in xyz
instring(xyz,",")→s1                 ©s1 is the position of the first comma
instring(xyz,",",s1+1)→s2            ©s2 is the position of the second comma

©Test for commas
if s1=Ø or s2=Ø then
 dialog                              ©Display error message if both commas
  title "ERROR"                      ©not found
  text "You must put commas"
  text "between x, y and z"
 enddlog
 if ok=Ø:return                      ©Let user quit by pressing [ESC],
 goto tryagain                       ©or try again by pressing [ENTER]
endif

©Convert x,y and z strings to expressions
expr(left(xyz,s1-1))→x
expr(mid(xyz,s1+1,s2-s1-1))→y
expr(right(xyz,dim(xyz)-s2))→z

©Just for a demo, display x, y and z. Your application probably won't do this
dialog
 title "SHOW X, Y AND Z"
 text "x is "&string(x)
 text "y is "&string(y)
 text "z is "&string(z)
enddlog

EndPrgm
```

If you need to use this method more than once in a dialog box, you can reduce the total code size by with a subroutine which converts a string of parameters to a list. These two functions, *rq2v()* and *rq3v()*, perform the conversion. Use *rq2v()* for two parameters:

```
rq2v(st)
Func
©Convert ("a,b") to {a,b}
©31marØ1/dburkett@infinet.com

local s

instring(st,",")→s

if s=Ø then
 return "ERR"
else
 return {expr(left(st,s-1)),expr(right(st,dim(st)-s))}
endif

EndFunc
```

Use *rq3v()* for three parameters:

```
rq3v(st)
Func
©Convert ("a,b,c") to {a,b,c}
©31marØ1/dburkett@infinet.com

local s1,s2

instring(st,",")→s1
instring(st,",",s1+1)→s2

if s1=Ø or s2=Ø then
 return "ERR"
else
 return
{expr(left(st,s1-1)),expr(mid(st,s1+1,s2-s1-1)),expr(right(st,dim(st)-s2))}
endif

EndFunc
```

Both of these functions return *"ERR"* if the commas are missing. This example shows how to use both functions.

```
©Initialize prompt variables
"Ø,Ø"→ab
"Ø,Ø,Ø"→cde

©Prompt for input variables
dialog
 request "a,b",ab
 request "c,d,e",cde
enddlog

©Extract a, b variables
util\rq2v(ab)→res
if res="ERR" then
 (... handle error here ...)
```

```
else
 res[1]→a
 res[2]→b
endif

©Extract c, d, e variables
util\rq3v(cde)→res
if res="ERR" then
  (... handle error here ...)
else
 res[1]→c
 res[2]→d
 res[3]→e
endif
```

*ab* and *cde* are strings that are returned from the *Request* commands in the dialog box. These are passed directly to *rq2v()* and *rq3v()*. The returned lists are saved in variable *res*. For example, if the users enters

1,2

at the *ab* prompt, and

3,4,5

at the *cde* prompt, then *ab* = "1,2" and *cde* = "3,4,5". The *res* result variable will be, respectively, {1,2} and {3,4,5}. During the variable extraction sections, the individual elements of *res* are stored to the appropriate variable. Note that both extraction sections test for the error condition. The actual error handling depends on your program. Typically, you would display an error message and give the user another chance to correctly enter the variables.

You can use this method to get numbers and expressions from the user, but not multiple lists or matrices, because the programs do not distinguish between the commas that separate the expressions and the commas within the expressions.

### [9.13] Disable ALPHA default in TI-89 AMS 2.05 dialog boxes

When a *Request* function in a dialog box prompts for input, AMS 2.05 automatically turns the Alpha input mode on. This 'feature' is not implemented on the TI-92+, which has an alpha keyboard. Some users find this irritating, because usually you are entering numeric parameters. So, you either have to remember to press [ALPHA] to disable the alpha keyboard, or you will type characters when you meant to type numbers. Fortunately, Kevin Kofler has written a program that disables this feature. The program is called *autoaoff()*, and you can get it here:

http://ti89prog.kevinkofler.cjb.net/

*autoaoff()* is a TSR (terminate and stay resident) program for the TI-89 which does not reduce the amount of RAM available for normal calculations. It requires a special program for un-installation, which is included in the distribution. The *readme.txt* file is thorough and detailed.

*(Credit to Kevin Kofler)*

## [9.14] Use keyboard programs to quickly change mode settings

The mode settings (shown with the [MODE] key) control fundamental aspects of calculator operation. Many of the tips in this list show that the proper mode setting is important for getting the right result. If you use your calculator for a variety problem types you will switch modes quite often, and this is tedious because several steps are needed to change a setting: press the [MODE] key, scroll down to the mode category, open the mode submenu, scroll to the right setting, change it, and press [ENTER] to finally make the change. One solution to this problem is to use the built-in keyboard program feature to change modes quickly with a few keystrokes. This is particularly useful with mode settings shown in the status line: angle units, auto/exact/approx and graph type. Since these are shown in the status line, you can quickly see that the correct mode is set. The keyboard programs are executed with [DIAMOND] [1] for *kbdprgm1()*, [DIAMOND] [2] for *kbdprgm2()*, and so on. You can have up to nine *kbdprgm()* programs.

In general, we will use a [DIAMOND] key combination to toggle each mode setting. For example, if I set up *kbdprgm1()* to cycle through the Auto/Exact/ mode, then repeatedly pressing [DIAMOND] [1] changes the setting through Auto, Exact, Approx, Auto and so on. If I use *kbdprgm2()* to change the angle mode, then pressing [DIAMOND] [2] changes the setting through RAD, DEG, RAD and so on.

We use *getMode()* to find the current mode, then *setMode()* to change the setting to the next option. The forms of *getMode()* and *setMode()* which we need are

```
getMode(modeString)

setMode(modeString,setString)
```

where *modeString* specifies the mode to get or set, and *setString* specifies the new setting. *modeString* and *setString* can be specified in one of two ways: either as text strings in the current language, or as numeric strings. I use numeric strings because this makes the programs independent of the current language, and it is easy to calculate the next mode setting.

We could used repeated *If ... then ... elseif ...* structures to change the mode, but any mode setting change can be done with one line of code:

```
setMode(modeString,string(exact(mod(expr(getMode(modeString)),n)+1)))
```

where *n* is the number of setting options. For example, if we want to change the Auto/Exact setting, then *modeString* = "14" and *n* = 3, since there are three choices. This statement changes the mode like this, working from the inside out:

Get the current mode as a string:
```
getMode(modeString)
```

Convert the mode string to a number:
```
expr(getMode(modeString))
```

Calculate the next mode setting number:
```
mod(expr(getMode(modeString)),n)+1
```

Use *exact()* to remove any decimal points or exponent characters:
```
exact(mod(expr(getMode(modeString)),n)+1)
```

Convert the next mode number to a string:
```
string(exact(mod(expr(getMode(modeString)),n)+1))
```

Change the mode setting:
`setMode(modeString,string(exact(mod(expr(getMode(modeString)),n)+1)))`

The next mode setting number is calculated with the *mod()* function which provides the wrap-around we need. For example, if there are three setting options, then we need to convert a 1 to 2, 2 to 3 and 3 to 1. The calculation works like this for n = 3:

mod(1,3)+1 = 2 *(setting 1 changes to setting 2)*
mod(2,3)+1 = 3 *(setting 2 changes to setting 3)*
mod(3,3)+1 = 1 *(setting 3 wraps around to setting 1)*

Applying *exact()* to the mode setting number calculation removes any decimal points or exponent characters, which may be present, depending on the current display format setting. This must be done because *setMode()* does not accept these characters in the *setString* argument. By including *exact()*, the expression will work with any display mode setting and in Approx mode.

This table shows the values of *modeString* and *n* for the modes which can be set with this method.

***modeString and number of options for various modes***

| Mode category | modeString | n |
|---|---|---|
| "Graph" | "1" | 6 |
| "Display digits" | "2" | 26 |
| "Angle" | "3" | 2 |
| "Exponential Format" | "4" | 3 |
| "Complex Format" | "5" | 3 |
| "Vector Format" | "6" | 3 |
| "Pretty Print" | "7" | 2 |
| "Split screen" | "8" | 3 |
| "Number of Graphs" | "11" | 2 |
| "Graph 2" | "12" | 6 |
| "Split screen ratio" | "13" | 2 for TI-89, 3 for TI-92+ |
| "Exact/Approx" | "14" | 3 |
| "Base" | "15" | 3 |

Refer to the *User's Guide* entry for the *setMode()* function for descriptions of the mode setting options.

Note that modes "Split 1 App" and "Split 2 App" do not use number codes for the settings, and the "Language" mode does not use a number code for the mode itself. These modes cannot be set with this method.

If you use several keyboard programs to change modes, you can save a little memory by using a program to change the mode settings:

```
modecycl(mode,n)
Prgm
©("mode",n) Toggle "mode" with n options
©3janØ2/dburkett@infine.com
setmode(mode,string(exact(mod(expr(getmode(mode)),n)+1)))
EndPrgm
```

then you could use these keyboard programs with [DIAMOND] [1], [DIAMOND] [2] and [DIAMOND] [3]:

```
kbdprgm1()
Prgm
©Change Angle setting
modecycl("3",2)
EndPrgm


kbdprgm2()
Prgm
©Change Exact/Auto/Approx setting
modecycl("14",3)
EndPrgm

kbdprgm3()
Prgm
©Change Graph setting
modecycl("1",6)
EndPrgm
```

Note that the [DIAMOND] keys are defined in the same order that the Mode indicators appear in the display status line. This makes it easier to remember the key which changes a particular mode.

For the Angle units, Auto/Exact and the Graph modes, these programs work well because the setting is shown in the status line. However, the other mode settings are not shown, so we don't know what the current setting is, nor what we have changed it to. In addition, some mode settings have so many options that it is not as helpful to cycle through them: it takes just as much time to do that as it does to use the built-in Mode screen.

With nine keyboard programs, it can be difficult to remember which programs are assigned to which keys. It helps to assign the keyboard programs in groups, as we did here, with some logical ordering to the key assignment.

### [9.15] [2nd] [ENTER] quickly selects, copies entries to command line

Push and hold [2nd], then repeatedly press [ENTER]. With each [ENTER] press, each entry in the history display is shown in the command line. When you release [2nd], the current entry remains in the command line. This isn't too surprising, considering that legend for the [2nd] function above the [ENTER] key is labelled ENTRY, but you might not have noticed this feature.

### [9.16] Use "Save Copy As" to display the current variable name in editors

Once you are editing a variable in one of the built-in application editors, the variable name is not displayed. For example, when you are editing a matrix, the matrix variable name is not shown. To display the variable name, press [F1] to display the Tools menu, then select "2:Save Copy As ...". You can select this option by highlighting it with the [UP] and [DOWN] keys, then pressing [ENTER], or by pressing [2]. You can also select this option by pressing [DIAMOND] [S], without pressing [F1]. In any case, once the "Save As" option is selected, the title of the current variable is shown in the dialog box title.

This method works with all the built-in editors: the Data/Matrix editor, the text editor, and the program editor.

*(from the TI-92 Plus FAQ)*

### [9.17] Speed up the keyboard

*(This tip is Francesco's submission to ticalc.org. I haven't tried it yet, so proceed at your own risk. I have made very minor editing changes. If you make even the most trivial mistake with this procedure, you may well completely hang up your calculator. You've been warned.)*

*Speed up the keyboard TI-89 / TI-92 Plus; mini how-to*
By Francesco Orabona
4/17/2002

*Introduction*

There are some TSR programs that speed up the keyboard of TI89/TI92+, but they have some problems:

- they have some bugs
- sometimes are old and not fully compatible with HW2
- they always need HW2Patch or h220xtsr, even if there isn't written in their docs!
- they need a little amount of memory
- some kernels will uninstall them without freeing the memory used.

For all these reasons I have thought to follow an alternative and, IMHO, simpler way: to change the values directly in the ROM! The steps I have followed are very simple.

*The steps*

Launch your favorite hex editor (for example I have used UltraEdit).

Open the ROM file *.89u/*.9xu. Go to the address

| | |
|---|---|
| TI-89 2.05: | 2CC9A and 2CC9B |
| TI-89 2.04: | 3330D and 3330E |
| TI-89 2.03: | 94153 and 94154 |
| TI-89 2.01: | 96CD6 and 96CD7 |
| | |
| TI-92 Plus 2.05: | 2CA80 and 2CA81 |
| TI-92 Pluse 2.01: | 965DC and 965DD |

The numbers 01 and 50 should be at these addresses. Change 01 to 00 and 50 to 64.

Go to the address

| | |
|---|---|
| TI-89 2.05: | 2CCBD |
| TI-89 2.04: | 33330 |
| TI-89 2.03: | 94176 |
| TI-89 2.01: | 96CF9 |
| | |
| TI-92 Plus 2.05: | 2CAA3 |
| TI-92 Plus 2.01: | 965FF |

The number 24 should be at this address. Change it to 14. Save the file.

Send the modified ROM to the calculator with TIBReceiver (You can find it on www.ticalc.org, read carefully its doc!) Note that you cannot use the last version of TI GraphLink to send a modified ROM, you must use another program, for example, I have used TilP.

Turn the calculator off, then on. Now the keys should be faster

All done! Simple! To be sure you have done the correct changes, before sending the ROM to the calculator, use VTI to verify the modified ROM, try also to turn off and on the emulated calculator.

*Technical explanation*

(If you want to know more read this paragraph)

A TSR program that speeds up the keyboard installs itself as a replacement of the trap #4: in this way every time the calculator is turned off and on, the new trap replaces the default values set by the AMS. I have searched in all the ROMs the addresses of these instructions: move.w #$150,something and move.w #$24,something. Then I have modified the values $150 and $24 with the ones set by the program Fast Keyboard.

*Credits*

Thanks to:

Jeff Flanigan for the program Fast Keyboard that has given me the idea
Kevin Kofler for his explanation on the problem of the TSRs on HW2

*Disclaimer*

Although this method has been tested, it is still remotely possible that it may do crash or hang your calculator. I am not responsible for any damage done to your calculator.

*Support*

Send me any suggestion, any ideas at bremen79@infinito.it and also tell me what you think of it.
www.genie.it/utenti/bremen79

*(Credit to Francesco Orabona)*

### [9.18] Use multiple commands on entry line with ":"

The colon separator ":" can be used to enter multiple commands or function calls on the entry line. For example this keystroke sequence

```
[3] [STO] [x] [:] [4] [STO] [y] [ENTER]
```

stores the values 3 and 4 to variables x and y, respectively. This doesn't save any keystrokes, but if you find yourself repeating the same sequence of commands, it can save time, as well as lines in the history display. Note that only the answer of the last command is shown in the history display. In the example above, 4 is shown.

You can also use this feature to write small programs on the command line. This contrived example shows a loop which creates a list of values:

```
For k,1,10:sin(k)→l[k]:EndFor:delvar k:l
```

Actually, it is easier to use the *seq()* function for this, but the example shows the basic idea. The loop creates a list of sin(x) for *x* from 1 to 10. The list is shown in the history display by the last *l* in the entry line. The global variable *k* is deleted with the command *delvar k*.

As a final example, suppose that you have a several sets of data, and you want to find the minimum and maximum residuals that result from fitting a straight-line regression to the data sets. If the data is saved in lists *a* and *b*, this entry-line sequence finds the results:

```
linreg a,b:regeq(a)-b→t:{min(t),max(t)}
```

For each regression, store the x- and y-data to lists *a* and *b*, then use the [UP] and [ENTER] keys to paste the expression to the entry line and execute it. Alternatively, you can use [2nd] [ENTRY] to choose and paste the expression.

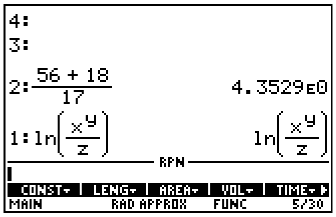
For expressions which generate large results on the history display, you can terminate the command with :0, then only 0 will be displayed. For example, to suppress the display of the 10 x 10 matrix created with *newMat()*, use

```
newMat(10,10):0
```

### [9.19] Use a more efficient RPN interface

The algebraic interface used on the TI-89/92+ is only one possible interface. One alternative is called RPN, which is an acronym for Reverse Polish Notation. RPN is popular with some users (including me) because it is more keystroke efficient than algebraic entry, and no parentheses are used. RPN is the method historically used on Hewlett-Packard calculators such as the HP-48 and many others. Lars Frederiksen has written a thorough RPN interface for the TI-89/92+ calculators.

RPN may seem foreign if you have never used it, but most people can use it effectively within about thirty minutes or so. In addition, most RPN users soon prefer it over algebraic entry. A typical RPN screen looks like this:



4:

3:

2: $\frac{56 + 18}{17}$ 4.3529E0

1: $\ln\left(\frac{x^y}{z}\right)$ $\ln\left(\frac{x^y}{z}\right)$

RPN

CONST▾ | LENG▾ | AREA▾ | VOL▾ | TIME▾ ▸
MAIN RAD APPROX FUNC 5/30

While this looks like the history display of the usual algebraic entry method, it actually shows the RPN stack. A stack is a data structure in which elements are pushed on from the bottom, and functions operate on expressions on the stack. The [ENTER] key pushes expressions on the stack. For example, the keystrokes to add 2 to 4 are [2] [ENTER] [4] [+], and this illustrates the origin of the acronym RPN. Polish notation is named in honor of the Polish mathematician Jan Lukasiewicz (1878 - 1956), who developed a syntax for representing any expression without parenthesis. In Lukasiewicz' 'Polish notation', the expression to add 2 and 4 is + 2 4. The 'reverse' in RPN comes from the fact that the operator comes after the operands, instead of before.

In the simple 2 + 4 example, there are no keystroke savings: both methods take four keystrokes. However, consider this example:

| | |
|---|---|
| (a+b)/(c+d) [ENTER] | *algebraic; 12 keystrokes* |
| a [ENTER] b + c [ENTER] d + / | *RPN; 9 keystrokes* |

While the keystroke savings are real, even more significant is the fact that intermediate results are displayed immediately, which leads to earlier troubleshooting of incorrect entries. In addition, there is no mental effort expended on matching parentheses, and there are no calculation errors from mismatched parentheses. RPN is as useful with symbolic expressions as it is with numeric ones.

RPN calculators also have various stack manipulation functions, such as swap, drop, dup and rotate, which make calculations even more efficient.

There can be no argument that learning RPN takes some effort. But that effort will be repaid in orders of magnitude by faster, more accurate results.

You can get Lars' RPN at Roberto Perez-Franco's site:

http://www.perez-franco.com/symbulator/download/rpn.html

RPN is fully documented, with plenty of examples to get you started.

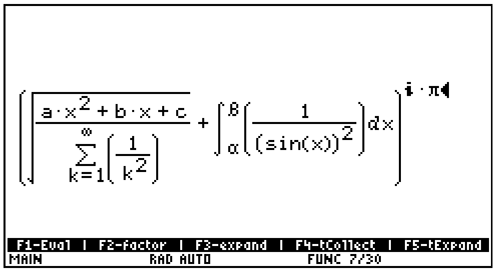
### [9.20] Use a graphical equation editor (equation writer)

Entering complicated expressions in the command line can be prone to error. Expressions can be entered more quickly without mistakes if you can see the equation, in pretty-print, as you enter it. This feature is called an equation writer, and has been available on the HP-48 and HP-49 calculators. E.W. has written an equation writer called EQW for the TI-89/92+. EQW is the winner of the 2001 Texas Instruments App Development Contest, 68000-platform division. You can get a free version at

http://tiger.towson.edu/~bbhatt1/ti/

or you can buy EQW as a flash application from TI. The flash app version has more features, but the free version is more than adequate for most uses.

This screen shot shows a typical EQW screen:



$$\left(\sqrt{\frac{a\cdot x^2+b\cdot x+c}{\sum_{k=1}^{\infty}\left(\frac{1}{k^2}\right)}}+\int_{\alpha}^{\beta}\left(\frac{1}{(\sin(x))^2}\right)dx\right)^{i\cdot\pi}$$

while the command line equivalent looks like this:

```
(√((a*x^2+b*x+c)/(Σ(1/k^2,k,1,∞)))+∫(1/(sin(x))^2,x,α,β))^(i*π)
```

EQW has these features:

- EQW is a no-stub ASM program.
- Edit expressions, equations, matrices and lists.
- Return expressions to the entry line, or to Lars Frederiksen's RPN program.
- Evaluate expressions and define functions without leaving EQW.
- Store and recall expressions
- Cut, copy and paste and UNDO are supported, as well as forms for frequently-used expressions
- Use function keys to factor, expand, collect, etc.
- Scroll large expressions
- Call EQW from your own TI Basic programs, and use dedicated [DIAMOND]-key functions
- Write extensions to enhance EQW with toolbar menus, pop-up menus and other programs.
- Only 24K bytes; can be archived.
- Complete documentation with examples

Steve Chism has written a Windows application, *TI Menus*, which automates the process of creating toolbar extensions for EQW. You can download *TI Menus* at *http://www.warp2k.com/eqlib/tbmaker.htm*

Nevin McChesney has written a sophisticated equation library program, *Equation Library*, which works with EQW. For more information, see *http://www.warp2k.com/eqlib/description.htm*

### [9.21] Programs for simple equation libraries

An equation library is a collection of equations together with a user interface to choose one of the equations, then solve for one of the equation variables. The HP-48 series has a substantial built-in library, and a similar library can be added to the HP-49G. The simple programs in this tip provide a simple but similar equation library for the TI-89/92+. These library programs do not include graphics, which are extremely helpful, and they cannot be used to solve multiple equations for a single variable.

Equation libraries are more useful for numeric solutions, not symbolic solutions. Once a symbolic solution is found, it is not necessary to find the same variable again. However, multiple numeric solutions may be required for equations in engineering and science applications. Therefore, the equation library programs use the built-in numeric solver. Using the built-in solver simplifies the programs, because the solver extracts the equation variables automatically. I also use the method of tip [7.10] to automatically start the numeric solver after the equation is chosen. The built-in solver uses the equation which is currently in the system variable *eqn*, and you can save any equation to that variable.

You provide the equation libraries. The library consists of a label (or name) for each equation, and the equation itself. It is easier to choose the equation by name, for example, *pendulum period*, than by choosing the equation. The library is a matrix with the equation names in the first column, and the equations in the second column.

If you have only a few equations, you can quickly choose a particular equation from one list of equation names.. But if you have dozens or hundreds of equations, it is more convenient to first choose a group of equations, then the desired equation from that group. To meet both requirements I show two equation library programs. *eqlib()* uses a single list of equations. *eqlibx()* uses groups of equations.

*Equation library with one menu*

The program for the single-menu equation library is:

```
eqlib()
Prgm
©Eqn library, one menu
©14novØ1/dburkett@infinet.com
©main\eqdef[n,2] holds labels & equations

local k

© Initialize pop-up menu index to zero, to allow use of [ESC] key to quit
Ø→k

© Extract and display equation labels as a pop-up menu
popup mat▸list(subMat(main\eqdef,1,1,rowDim(main\eqdef),1)),k

© Quit if [ESC] pressed: no equation chosen
if k=Ø:return

© Set the equation and start the numeric solver
main\eqdef[k,2]→eqn
setMode("Split 1 App","Numeric Solver")

EndPrgm
```

This is a very simple equation library, but it can be implemented in only five functional lines of TI Basic.

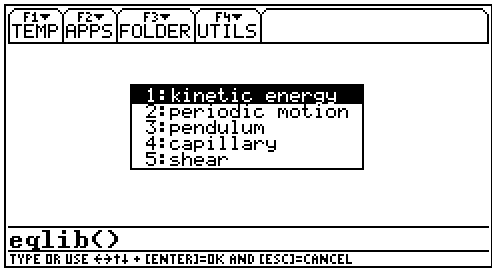
Before you can use *eqlib()*, you must set up the equation library in matrix *main\eqdef*. You can create this matrix in the entry line, or you can use the built-in matrix editor by pressing [APPS] [6] [3] to create a new matrix. Refer to the complete TI-89/92+ user's guide to learn how to use the matrix editor. As an example, *eqdef* might look like this:

|   | c1 | c2 |
|---|---|---|
| 1 | "kinetic energy" | .5*m*v^2=e |
| 2 | "periodic motion" | x=a*cos(ω*t+ϕ) |
| 3 | "pendulum" | t=2*π*√(l/(d*g*m)) |
| 4 | "capillary" | h=2*cos(α)*τ/(g*ρ*r) |
| 5 | "shear" | f*tan(θ)/(a*g)=d |

The first column *c1* holds the equation labels. These are the names that are shown in the pop-up menu to choose the equation, so use names such that you can remember the equation to which they apply. The names are strings with double-quote characters. The second column *c2* holds the equations themselves.

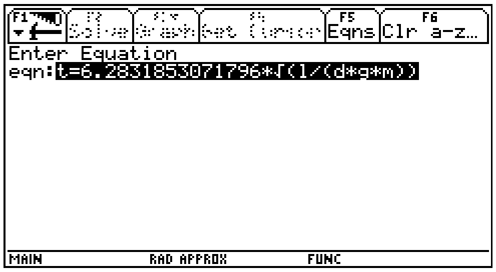
If you later need to edit your equation library, it is easily done in the matrix editor. You can use the [F6] Util menu to insert and delete rows.

When you start *eqlib()*, this pop-up menu is shown:



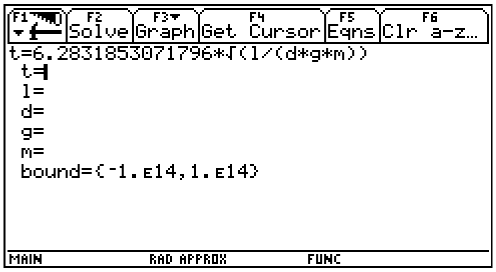
```
eqlib()
TYPE OR USE ←→↑↓ + [ENTER]=OK AND [ESC]=CANCEL
```

You can press [ESC] at this point to exit the equation library and return to the home screen. If you press [3] to choose the pendulum equation, the numeric solver is started:



```
Enter Equation
eqn:t=6.2831853071796*√(1/(d*g*m))

MAIN          RAD APPROX          FUNC
```

Press [ENTER] to accept this equation, and the numeric solver continues:



```
t=6.2831853071796*√(1/(d*g*m))
 t=
 l=
 d=
 g=
 m=
 bound={-1.E14,1.E14}

MAIN          RAD APPROX          FUNC
```

Note that the equation variables are displayed. Refer to the complete user's guide to learn how to use the numeric solver. In general, use [UP] and [DOWN] to choose the known variables and enter their values, then select the unknown variable and press [F2] Solve, to find the solution. The numeric solver works more quickly if you edit the *bound* variable list, and supply an initial guess for the unknown variable. For example, if you know that the solution is positive and less than 1000, you might set *bound* to {0,1000}. If you know that the solution should be about 120, then you would set the unknown variable to 120 before pressing [F2] to solve for it.

Unfortunately *bound* cannot be set from TI Basic. If this was possible, we could further automate the solution process by including a default *bound* setting in the equation library definition matrix.

*Equation library with multiple menus*

If your often use many equations, it is awkward to choose the correct equation from the single pop-up menu used in *eqlib()*. One solution is to use multiple equation library matrices to organize groups of equations. *eqlibx()*, shown below, implements this idea.

```
eqlibx()
Prgm
©Eqn library, multiple menus
©14novØ1/dburkett@infinet.com
©main\eqlxmenu{} holds menu names
©main\eqdef{m}[n,2] holds labels & equations; {m} = 1,2,3,...

local k,n

Ø→k                                                 © Display 'menu of menus'
popup main\eqlxmenu,k
if k=Ø:return                                       © Quit if [ESC] pressed

"main\eqdef"&string(exact(k))→n                     © Build name of library matrix

Ø→k                                                 © Display equation menu
popup mat▸list(subMat(#n,1,1,rowDim(#n),1)),k
if k=Ø:return                                       © Quit if [ESC] pressed

#n[k,2]→eqn                                         © Set chosen equation
setMode("Split 1 App","Numeric Solver")             © Start numeric solver

EndPrgm
```

This program displays two pop-up menus. The first menu shows the names of the different libraries. The second menu shows the equations in the chosen library. The library names are stored in the list variable *main\eqlxmenu*. For example, you might have three groups of equations for geometry, electricity, and physics, and you could use this for *main\eqlxmenu*:

```
{"geometry","electricity","physics"}
```

There is a corresponding equation matrix for each element of *eqlxmenu*, and the matrices have specific names. The library for the first element is *main\eqdef1*, for the second element is *main\eqdef2*, and so on. For our example, the equation matrices are

| | |
|---|---|
| geometry: | *main\eqdef1* |
| electricity: | *main\eqdef2* |
| physics: | *main\eqdef3* |

The format of these equation matrices is the same as that of *eqlib()* above: the first column *c1* is the equation label, and the second column *c2* is the equation. These examples show typical equation matrices.

***Matrix main\eqdef1 for geometry equations***

| | c1 | c2 |
|---|---|---|
| 1 | "cosines law" | `c^2=a^2-2*a*b*cos(cc)+b^2` |
| 2 | "triangle area" | `.5*a*b*sin(cc)=area` |
| 3 | "polygon area" | `tan(π/k)*k*r^2=area` |

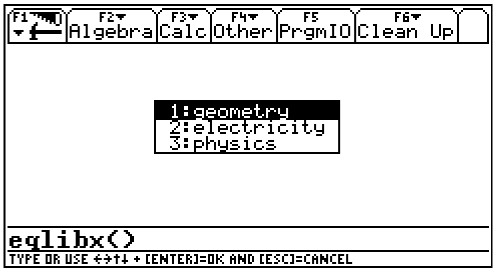
***Matrix main\eqdef2 for electricity equations***

|   | c1 | c2 |
|---|---|---|
| 1 | "Ohm's law" | v=i*r |
| 2 | "voltage divider" | vo=v*(rr2/(rr1+rr2)) |
| 3 | "parallel resistance" | r=(rr1*rr2)/(rr1+rr2) |

***Matrix main\eqdef3 for physics equations***

|   | c1 | c2 |
|---|---|---|
| 1 | "kinetic energy" | .5*m*v^2=e |
| 2 | "periodic motion" | x=a*cos(ω*t+ϕ) |
| 3 | "pendulum" | t=2*π*√(l/(d*g*m)) |

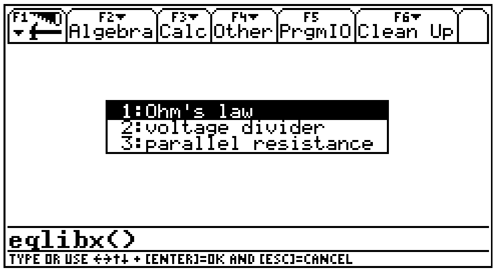
When *eqlibx()* is executed, the first menu is shown:



eqlibx()

TYPE OR USE ←→↑↓ + [ENTER]=OK AND [ESC]=CANCEL

If we press [2] to choose the 'electricity' menu item, a second menu is shown, from which the desired equation is selected:



eqlibx()

TYPE OR USE ←→↑↓ + [ENTER]=OK AND [ESC]=CANCEL

After the equation is selected, the numeric solver is started, and you can set and solve for the variables as needed.

### [9.22] Minimize TI-89 alpha entry inefficiencies

The TI-89 alpha characters share keys with other functions. The characters are in alphabetic order except that [X], [Y], [Z] and [T] are dedicated keys. Some users find it very tedious to key in alphanumeric entries such as function and variable names. These tips may make alpha entry more efficient.

- Learn the keyboard. While this seems obvious, some users become quite proficient.
- Use [CUT], [COPY] and [PASTE] to reenter variable and function names.
- Use the [MATH] and [CATALOG] menus to enter function names. Use [F4] in the CATALOG display to enter user function and program names. Variable and function names can also be entered from the VAR-LINK menu.
- Use [ANS], as well as [UP], [DOWN] and [ENTER] to retrieve entries and answers from the history display, then edit them in the command line.
- Use simple variable names, especially those with "x", "y", "z" and "t". Remember, though, that y1 - y99, z1 - z99 and t0 are reserved system variable names and cannot be used.
- Take advantage of the custom toolbar menus. The default toolbar menu set includes many common commands, but using your own menus, with the commands you frequently need, will save even more time. Refer to other tips for ways to enhance custom menus.
- Consider assigning shifted-key characters to custom toolbar menu items. For example " and " is useful with *solve()*. You may even want to make menu entries for "{}" and "[]" to speed list and matrix entry.
- Remember that you can have up to nine 'keyboard programs' which are executed with [DIAMOND] [1] to [DIAMOND] [9]. These are particularly effective when each keyboard program is assigned to a different custom toolbar menu.
- Some users prefer to hold [alpha] while typing characters. This can be more convenient than using the alpha-lock ([a-lock]) key.

# 10.0 Units Conversion Tips

### [10.1] Units calculations are approximate in Auto mode

Bhuvanesh reports:

*When in AUTO or APPROX mode, the units system works in approximate mode. So, for example,*

```
9*_dollars/_hour*8*_hour/_day*5*_day/_week*12*_week/_summer
```

*returns*

```
432Ø.ØØØØØØØØØ2*(_dollars/_summer)
```

*Note that exact(ans()) still returns this answer. If this calculation is performed in EXACT mode, it returns*

```
432Ø.*(_dollars/_summer)
```

*Notice that it keeps the decimal point, even in EXACT mode.*

*(credit to Bhuvanesh Bhatt)*

### [10.2] Convert compound units to equivalent units

In some cases, you can use the built-in units conversion to express some units in terms of base units. Some examples:

```
_coul▸_s            1·_A·1·_s

_J▸_N               1·_m·1·_N

_F▸1/_V             1·_coul/1·_V

_W▸1/_s             1·_J/1·_s

_henry▸1/_s         (1·_kg·1·m²) / (1·_A·1·_coul·1·_s)

_Wb▸1/_A            1·_J / 1·_A
```

If the converted-to unit appears in the denominator of the result, you need to express that unit as a reciprocal in the conversion command.

This method does not necessarily return the converted unit in terms of base units. This can be seen in the third example above, in which capacitance in farads (F) is converted to coulomb/volt. Neither the coulomb nor the volt are bases units; a true base-units conversion would be ($A^2$*$s^4$)/(kg*$m^2$). For reference, the seven SI base system units are:

Length, meter (m)
Mass, kilogram (kg)
Time, second (s)
Electric current, amp (A)
Temperature, degrees Kelvin (K)
Luminous intensity, candela (cd)
Amount of substance, mole (mol)

If a conversion includes units other than these seven, it is not a base unit conversion.

*(credit to anonymous poster)*

### [10.3] Remove units from numbers

Suppose *x* is a number with an associated unit, for example, 1.2_m. To extract just the numeric value, use

```
part(x,1)
```

To extract just the units, use

```
part(x,2)
```

For example,

```
part(1.2_m,1)          returns         1.2
part(1.2_m,2)          returns         1*_m
```

This also works for compound units, for example

`part(1.2_m*_s/_mol,2)` returns $\frac{1\cdot\_m\cdot\_s}{1\cdot\_mol}$

### [10.4] Add units to undefined variables

Units can be applied to variables and expressions, as well as to numbers. For example, to create a variable *tf* with units of °F, use

```
tf*_°F
```

The important point is to explicitly multiply the variable name by the units. This is not necessary when applying units to numbers. A *domain error* message results if you do not use the explicit multiplication.

The example above gives different results, depending on the value of *tf*:

| | |
|---|---|
| If *tf* is undefined: | `tf·_°F` |
| If *tf* is a number, like 12: | `12·_°F` |
| If *tf* is another variable name, like *tf2*: | `tf2·_°F` |

### [10.5] Use *tmpcnv()* to display temperature conversion equations

*tmpcnv()* usually performs temperature conversion on numbers, for example

```
tmpcnv(Ø_°C,_°F)
```

results in 32_°F. You can also use *tmpcnv()* to display the temperature conversion equations, if you convert an undefined variable, like this:

```
tmpcnv(TF*_°F,_°C)
```

which gives the result

```
Ø.5556·(TF-32)·_°C
```

### [10.6] Add units to built-in Units menu

The built-in 89/92+ Units menu is lacking many common units. For example, the Area menu has only two units, and the Accelration menu has no units. You can remedy this by defining your own units, in terms of the built-in units. In addition, the 89/92+ can recognize the categories for many of these units, and place them in the correct category so you can access them with the [UNITS] key.

The basic idea is to store the unit definition as a user-defined unit. For example, to save a cubic inches unit as a unit called _in3, use

```
_in^3→_in3
```

Note the use of the underscore character '_', which defines a unit. Now the Volume units menu shows

```
...
_galUK
_in3
_l
...
```

This unit can be used just like any other built-in unit. For example, to convert 5 liters to cubic inches, use

```
5_l▸_in3
```
which returns 305.12_in3

If you use many user-defined units, it is convenient to have a program that defines them all at once. Here is an example of such a program:

```
units()
Prgm
©Define new units
©1Øoct ØØ dburkett@infinet.com

©Area units
_in^2→_in2
_ft^2→_ft2
_yd^2→_yd2
_mi^2→_mi2
_cm^2→_cm2
_m^2→_m2
_km^2→_km2

©Volume units
_in^3→_in3
_ft^3→_ft3
_yd^3→_yd3
_mi^3→_mi3
_cm^3→_cm3
_m^3→_m3
_km^3→_km3
```

```
    ©Speed units
    _ft/_s→_fps
    _m/_s→_mps

    ©Acceleration units
    _ft/_s^2→_fps2
    _m/_s^2→_mps2

    ©Density units (not in built-in menu)
    _kg/_m^3→_kgm3
    _lb/_ft^3→_lbft3

    EndPrgm
```

Just run *units()* to define all the units it includes. Note that the density units will not be put in the built-in menus, since there is no Density category. These units can be used by typing them in the command line, or by putting them in a custom menu.

Note that your custom units cannot be archived. The 89/92+ considers them to be system variables, and prevents the archive.

*(Credit to Mike Grass)*

# 11.0 Solving Tips

### [11.1] Try *nsolve()* if *solve()* and *csolve()* fail

*nsolve()* may be able to find a solution when *solve()* and *csolve()* cannot. For example, these both return *false*:

```
cSolve((x+1)^(x+2)=0,x)
Solve((x+1)^(x+2)=0,x)
```

However,

```
nSolve((x+1)^(x+2)=0,x)
```

returns a solution of x = -1. As usual, all solutions should be checked by substituting the solution into the original equation.

*(credit to Bhuvanesh Bhatt)*

### [11.2] *zeros()* ignores constraint for complex solutions

In some cases the *zeros()* function will ignore constraints that require a complex solution, and return a general solution instead of False. For example:

$$\text{zeros}\left(\frac{x^2-xy+1}{x},x\right)\,|\,|y|<1$$

returns two solutions

$$\left\{\frac{\sqrt{y^2-4}+y}{2}\quad\frac{-\left(\sqrt{y^2-4}-y\right)}{2}\right\}$$

Since both of these solutions return complex results for the original constraint |y| < 1, *zeros()* should really return False.

*(Credit to Bhuvanesh Bhatt)*

### [11.3] Try *cZeros()* and *cSolve()* to find real solutions

*zeros()* and *solve()* may return no solutions even when real solutions exist, for example:

```
solve((-1)^n=1,n)
```

returns *false*, and

```
zeros((-1)^n-1),n)
```

returns {}, indicating that no solution was found. However,

```
cSolve((-1)^n=1,n)
```

returns n = 2*@n1, and

```
cZeros((-1)^n-1,n)
```

returns {2*@n1}; both of these results are correct.

As another example, consider

```
cSolve(ln(e^z_)=ln(z_^2))
```

As shown, with no guess, the returned result is

```
e^z_-z_^2=Ø
```

which doesn't help much. However, with an initial complex solution guess:

```
cSolve(ln(e^z_)=ln(z_^2),{z_=i})
```

*cSolve()* returns one complex result of 1.588... + 1.540...i. With a real guess:

```
cSolve(ln(e^z_)=ln(z_^2),{z_=Ø})
```

*cSolve()* returns one real result of -0.7034... Note that *cSolve()* returns these approximate floating-point results even in Exact mode.

*(Credit to Bhuvanesh Bhatt)*

### [11.4] Using *solve()* with multiple equations and solutions in programs

This rather involved tip relates to using the *solve()* function in programs, when multiple equations to be solved are generated within the program. The basic idea is to build a string of the equations to be used in the *solve()* function, evaluate the solve function string with *expr(),* then change the results into a more useful list form. This tip only applies if you are interested in numeric solutions, not symbolic solutions.

As an example, I'll use this system of equations to be solved:

$ax_1^2 + bx_1 + c = y_1$
$ax_2^2 + bx_2 + c = y_2$
$ax_3^2 + bx_3 + c = y_3$

where the unknown variables are *a*, *b* and *c*. $x_1$, $x_2$, $x_3$, $y_1$, $y_2$ and $y_3$ are given. Actually, since this is a linear system, you wouldn't really use *solve();* instead you would use the matrix algebra functions built into the 89/92+. But it makes a good example.

Suppose that we know that

| | |
|---|---|
| $x_1 = 1$ | $y_1 = -1$ |
| $x_2 = 2$ | $y_2 = -11$ |
| $x_3 = 3$ | $y_3 = -29$ |

so the equations to be solved are

a + b + c = -1
4a + 2b + c = -11
9a + 3b + c = -29

To make the eventual routine more general-purpose, rewrite the equations to set the right-hand sides equal to zero:

a + b + c + 1= 0
4a + 2b + c + 11 = 0
9a + 3b + c + 29 = 0

This routine, called *solvemul(),* returns the solutions as a list:

```
solvemul(eqlist,vlist)
func
©Solve multiple equations
©eqlist: list of expressions
©vlist: list of variables
©returns list of solutions in vlist order
©calls strsub() in same folder
©25dec99/dburkett@infinet.com

local s,vdim,k,t,vk,vloc,dloc

dim(vlist)→vdim

©Build expression to solve
""→s
for k,1,vdim-1
 s&string(eqlist[k])&"=Ø and "→s
endfor
s&string(eqlist[vdim])&"=Ø"→s

©Solve for unknown variables
string(expr("solve("&s&","&string(vlist)&")"))→s

©Convert solution string to list
newlist(vdim)→t

strsub(s,"and",":")→s ©Change "and" to ":"
strsub(s," ","")→s ©Strip blanks

for k,1,vdim
 instring(s,string(vlist[k]))→vloc
 instring(s,":",vloc)→dloc
 if dloc=Ø: dim(s)+1→dloc
 mid(s,vloc+2,dloc-vloc-2)→t[k]
endfor

©Return coefficient list
seq(expr(t[k]),k,1,vdim)

Endfunc
```

The input parameters are *eqlist* and *vlist*, where *eqlist* is a list of expressions to solve, and *vlist* is the list of variables for which to solve. *solvemul()* assumes that the expressions in *eqlist* are equal to zero.

The routine returns the solutions in *vlist* order. Note that an external function *strsub()* is called. *strsub()* must be in the same folder as *solvemul().* See tip [8.2] for details.

To use *solvemul()* to solve our example, store two variables:

*eqs1* is {a+b+c+1, 4*a+2*b+c+11, 9*a+3*b+c+29}

*vars1* is {a,b,c}

then call *solvemul()* like this:

```
solvemul(eqs1,vars1)
```

and it returns

{-4, 2, 1}

which means that a = -4, b = 2 and c = 1, which happens to be the correct answer.

Again, there is no advantage to using this function to solve equations at the command line, or even equations in a program, if the equations are known in advance. This type of program is necessary when your application program generates the functions to be solved, and you don't necessarily know what they are before you run the program. Further, *solvemul()* returns the solutions as a list, which your program can use in further calculations, or display, as appropriate.

If you've read this tip list, word for word, up to this point, the first three parts of the function hold no surprises. The two input lists are built into a string that can be used in *solve(),* and then *expr()* is used to evaluate the string and solve for unknown variables.

However, the result of *solve()* is returned as a string that looks something like this:

"a=-4.0 and b = 2.0 and c=1.0"

The last parts of *solvemul()* convert this string to a list. First, I convert all the "and" occurrences to ":", and delete all the spaces, for a string like this:

"a=-4.0:b = 2.0:c=1.0"

Next, I loop to search the string for the variables in *vlist*, and extract the values for these variables. Using the ":" character as a delimiter makes this easier. Finally, I build the string of these result and return it.

To use this routine, you have to be sure that the solution string consists of only a single numeric solution for each unknown variable, so that the solution string does not have any "or"s in it. It would be easy to add a test for this error condition, and return an error code, but my example doesn't show that.

Further, a real application should test to ensure that *solve()* really did return valid solutions.

### [11.5] Using bounds and estimates with *nsolve()*

*nsolve()* is a function that is used to find one approximate real root of an equation. *nsolve()* is better than *solve()* in some cases, because *nsolve()* returns a single number, rather than an expression such as "x=4". This means that the result of *nsolve()* can be used in further calculations without parsing the number from the expression. Further, *nsolve()* is usually faster than *solve().*

The function format is

```
nsolve(equation,VarOrGuess)
```

where *equation* is the equation to be solved, and *VarOrGuess* specifies the variable for which to solve, and, optionally, a guess of the solution. As described in the manual, the command can be further modfied with solution bounds like this:

```
nsolve(equation,VarOrGuess)|bounds
```

where *bounds* is a conditional expression that specifies an interval over which to search for an answer. See the manual for examples.

Like any numeric solver, *nsolve()* can be faster if you supply a guess or bound the solution interval. Note that the guess need not a simple number, but can be a function itself.

With some functions, you *must* supply bounds to get the right solution. Consider y = $x^2$ as a simple example. If y = 4, then there are two solutions, x = 2 and x = -2. Since both are correct, and *nsolve()* cannot 'know' which solution you want, you need to supply the bound like this:

```
nsolve(x^2=4,x)|x>Ø
```

to find the x = 2 root

or

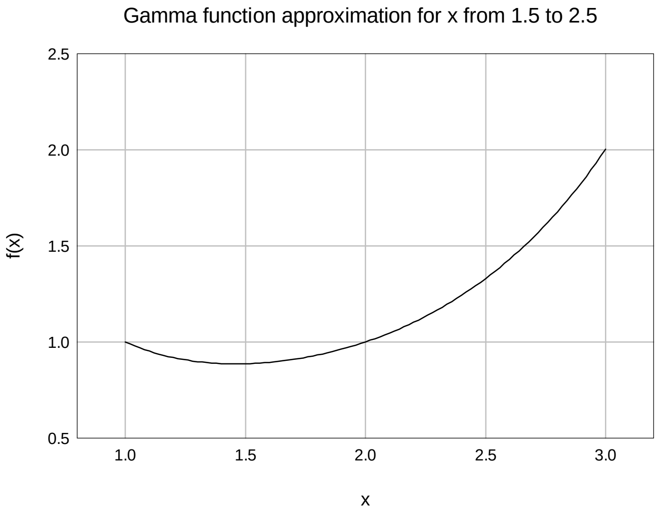
```
nsolve(x^2=4,x)|x<Ø
```

to find the x = -2 root.

To test the performance of the numeric solver, I found an estimating polynomial for the gamma function over the range x = [1.5, 2.5]. The function is

$$f(x) = a + bx + cx^2 + dx^3 + ex^4 + fx^5 + gx^6 + hx^7 + ix^8$$

This graph shows the function approximation.



First, note that we must use bounds to limit the solution range, because the function has a minimum at about x = 1.4. Otherwise, if we try to solve for f(x) = 1, we may get the solution x = 1 or x = 2.

I wrote a simple test program to solve for x for 11 different values of f(x) over the range [1.5, 2]. I also found an estimating function to find a guess for the solver. The estimating function is a 5th order polynomial, with an error of about ±0.03. The *nsolve()* call to use the estimating function looks like this:

```
nsolve(polyeval(fclist,xx)=yy,xx=polyeval(fglist,yy))
```

Here, *fclist{}* is the list of the 8th-order function polynomial coefficients, and *fglist{}* is the list of the 5th-order estimating function coefficients.

I tested three different conditions:

1. No initial guess and bounds of [1.48, 2.52]: mean execution time = 4.3 seconds.
2. Initial guess function and same bounds as 1.: mean execution time = 5.4 seconds
3. Initial guess function, no bounds: mean execution time = 4.2 seconds.

The error for all three conditions was the same. While this is not an exhaustive test, it shows the same results that I have often seen when using the solver:

- *nsolve()* is very good at finding the correct solution, when only bounds are given.
- Supplying *both* bounds and an initial guess function can actually result in slower execution time.
- In terms of execution time, supplying just bounds is as good as supplying an estimating function.

### [11.6] Use *solve()* as multiple-equation solver

The HP48/49 series of calculators have a useful feature called the 'multiple equation solver'. This is *not* the same as solving simultaneous equations. Instead, the equations have some variables in common, but not every equation has every variable. For example, consider the equations for Ohm's law and electrical power:

V = I * R
P = I * V

In this case, there are four variables, but by specifying any two, it is possible to solve for the remaining two variables. You can use the *solve()* function to simulate the HP48 multiple equation solving capability, like this:

```
solve(v=i*r and p=i*v,{r})|v=4.2 and p=1Ø
```

In this example I know v = 4.2V and p = 10 watts, and I want to find the resistance *r*. I can't find the resistance directly from either equation; I need both equations to find it. In this case, r = 1.764 ohms.

It is nearly as easy to find both unknown variables at once:

```
solve(v=i*r and p=i*v,{r,i})|v=4.2 and p=1Ø
```

which returns

r = 1.746 and i = 2.381

### [11.7] Saving multiple answers with *solve()* in program

*solve()* returns multiple answers for simultaneous equations. The answers are returned as an expression, for example

x1=1.234 and x2=5.678

This works well from the command line, but not within a program. For example,

solve({equation1 and equation2},var1,var2)→res1 [DOESN'T WORK!]

doesn't work, because TI Basic interprets the *solve()* result as a boolean, evaluates the boolean and saves the result; either *true* or *false*.

This works:

```
string(solve(...))→res1
```

This saves the results as a string, which can be displayed, or decomposed into individual answers by using *inString()* to search for the location of "and", then using *mid()* to extract the answers.

Another solution, which is better in most cases, is to use the *zeros()* function, which is equivalent to

```
exp▸list(solve(expression=Ø,var))
```

This returns all the solutions as a list, and the individual solutions can be extracted by their position in the list.

### [11.8] Try *solve()* for symbolic system solutions

Consider this system of equations:

2x - y + z = a
x + y - 3z = b
3x - 2z = c

The solution for x, y and z depends on the values of a, b and c. *solve()* can usually find solutions to these kinds of systems. For example,

```
solve(2x-y+z=a and x+y-3z=b and 3x-2z=c,{x,y,z})
```

returns

$$x=\frac{c+2\cdot @1}{3} \text{ and } y=\frac{3\cdot b-c+7\cdot @1}{3} \text{ and } z=@1 \text{ and } a=-(b-c)$$

The notation "@1" indicates an arbitrary constant in the solution. For example, let @1 = k, then

$$x=\frac{c+2\cdot k}{3} \text{ and } y=\frac{3\cdot b-c+7\cdot k}{3} \text{ and } z=k \text{ and } a=-(b-c)$$

If we let k = 5, b = 3 and c = 7, then a = 4, x = 17/3, y = 37/3 and z = 5. These values are solutions to the original system.

*(Credit to Rick Homard)*

### [11.9] *nSolve()* may return "Questionable Accuracy" warning even with good solutions

*nSolve()* is a reasonably robust numeric solver, but some equations give it difficulty, even when there are no obvious problems such as singularities. Futher, *nSolve()* may give the "Questionable Accuracy" warning in the status line, for solutions that seem valid. In these cases, *nSolve()* will take longer to return the answer, even when it returns nearby solutions quickly.

These two programs illustrate the problem:

```
solvebug(vf)
func
local vfs,tapx

©Find estimate tapx for solution
ln(1ØØØ*(vf-.Ø15))→vfs
polyeval({3.618919682914,⁻31.ØØ383Ø334444,76.7374729786Ø3,⁻68.2372Ø1523917,262.4
6139741751,84.9166293Ø6139},vfs)+polyeval({⁻3.9287348339733ᴇ⁻7,5.9179552Ø41553ᴇ⁻
5,⁻Ø.ØØ3689615561Ø467,Ø.123Ø899Ø642Ø18,⁻2.756Ø332337Ø98,Ø},1/vfs)→tapx

©Find solution, with bounds from tapx
nsolve(vfts12Ø(t)=vf,t)|t≥tapx-.4 and t≤min({tapx+.4,7Ø5.47})

Endfunc


vfts12Ø(ts)
```

```
func

polyeval({6.177102005557ᴇ⁻14,⁻3.724921779392ᴇ⁻11,9.3224938547231ᴇ⁻9,⁻1.239459227
407ᴇ⁻6,9.2348545475962ᴇ⁻5,⁻.0036542400520554,.075999519225692},ts)

Endfunc
```

*solvebug()* is a program that tries to find a solution *t* to the function vfts120(t) = vf, where *vf* is supplied as an argument. *vfts()* calculates a 6th-order polynomial polynomial function of its argument *ts*. Some typical values of vfts120(ts) near ts = 100 are

| ts | vfts120(ts) |
|---|---|
| 100 | 0.01612996896145 |
| 100.5 | 0.01613163512705 |
| 101 | 0.01613331366868 |

One way to test *solvebug()* solutions is to use a call like this:

```
solvebug(vfts120(t))
```

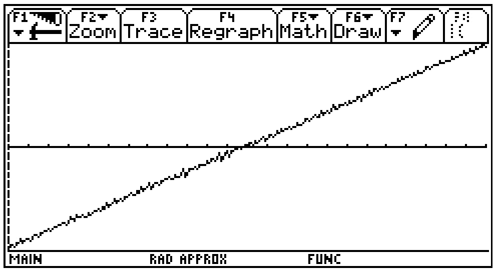
In general this expression should return a value near *t.* For example, if t = 100, this expression quickly returns 100. But if we set t = 100.17, it takes longer to find the solution, and the "Questionable Accuracy" warning appears in the display status line. However, the residual error is only about -6.6E-9, so the solution is as good as can be expected with *nSolve().*

*vfts120()* is well-behaved in this area, and very nearly linear. I emailed TI-cares about this problem, and here is their response:

> *"The evaluation of solvebug(vfts120(100.17)) results in the computation of nSolve(vfts120(t) = vfts120(100.17),t) with computed bounds that keep the search in the neighborhood of t = 100.17. On the Y= screen define y1(x) = vfts120(x) - vfts120(100.17). On the Window screen set the following window dimensions*
>
> *xmin = 100.17 - 1.2E-7*<br>
> *xmax = 100.17 + 1.2E-7*<br>
> *xscl = 1.E-8*<br>
> *ymin = -4.E-13*<br>
> *ymax = 4.E-13*<br>
> *yscl = 1.E-14*<br>
> *xres = 1.*
>
> *Graph y1(x). The graph shows a greal deal of roundoff noise due to catastrophic cancelation. This roundoff noise can also be seen by generating the Table corresponding to the Graph.*
>
> *The roundoff error causes several apparent local maximums in the neighborhood of the reported solution. These local maximums get very close to the x-axis; that is, they almost, but don't quite, produce sign changes. To the nSolve algorithm these apparent local maximums so close to zero appear to be "candidate" even-order solutions.*
>
> *Floating point roundoff error can perturb a legitimate even-order solution so that the residual doesn't change sign and doesn't quite reach zero. When such a residual is very-nearly zero, nSolve reports the position as a "candidate" solution but also issues the "Questionable accuracy" warning. The computed bounds affect the sequence of sample values, so they can strongly affect which solution or "candidate" solution is approached and the number of steps (length of time) to settle on that solution. Since nSolve only seeks one solution, it may report a "candidate" solution even when there is a nearby solution that DOES have a sign change.*

*To summarize, the computed bounds cause the sequence of nSolve steps to take a somewhat longer time to settle on a "candidate" solution that is produced by roundoff noise due to catastrophic cancelation. While this "candidate" solution is not the best available, it is a good approximation with a very small relative error. Given the catastrophic cancelation and the fact that the slope of the curve is extremely small in the neighborhood of the solution, the reported "candidate" solution is a very good result despite the conservative "Questionable Accuracy" warning.  Moreover, the computing time is quite modest for such an example."*

I followed TI's advice and plotted the function, which looks like this:



As TI wrote and this plot shows, the difference between vfts120(x) and vfts120(100.17) is not a smooth curve. The 'catastrophic cancellation' to which TI refers is also called destructive cancellation, and refers to computation errors that result from subtracting two nearly equal numbers. This results in a loss of significant digits during the calculation.

Note also that the y-scale for this plot is very small, ±4E-13, so the plot shows effects that are not usually evident with the display resolution of 12 significant digits.

In summary, be aware that *nSolve()* may return the "Questionable Accuracy" warning even for solutions that are fairly good. And, in situations like this, *nSolve()* will take slightly longer to find the solution.

### [11.10] *solve()* may return false solutions for trigonometric expressions

Executing the *solve()* function as shown:

```
solve(tan(x-1)/(x-1)=Ø,x)
```

returns

```
x = @n1*π + 1
```

The 89/92+ use the notation @n1 to indicate an arbitrary integer, which usually includes 0.  In this case, x = 1 is not a solution, since the expression is undefined at x = 1.

*solve()* does not even return the correct result for the limit as x approaches 1, since

$$\lim_{x\to 1}\frac{\tan(x-1)}{x-1}=1$$

This effect is also shown if *tan()* is replaced with *sin().*

*(Credit to Timité Hassan)*

### [11.11] *exp▸list()* fails with *solve()* in user functions

If entered from the command line, this expression

```
exp▸list(solve(x^2+x=Ø,x),x)
```

correctly returns the solutions {-1, 0} as a list. However, if you include this expression in a user function, the empty list {} is returned instead of the correct solution list. The fix to this problem is to use *expr()* to evaluate the expression as a string, as shown for example in this function:

```
t(xx,yy)
Func
 expr("exp▸list(solve(xx,yy),"&string(yy)&")")
EndFunc
```

*(Bug found by Michal Letowski; fix by Bhuvanesh Bhatt)*

### [11.12] *nSolve()* may ignore solution constraints

This tip demonstrates two cases for which *nSolve()* ignores solution constraints and returns a solution outside of the constraints. The first case involves the ordering of the conditional constraint if it uses local variables, and the second case relates to whether or not the independent variable is declared local or not. AMS 2.05 is used for the examples, but earlier AMS versions may also exhibit the problems.

*Conditional constraint ordering*

When used in a program or function, *nSolve()* does not always honor solution constraints if the constraints are local variables. If the constraint is expressed like this:

```
...
local low,high
value_1→low
value_2 →high
nsolve(f(x)=Ø,xØ|x>low and x<high →result
...
```

then *nSolve()* seems to return the correct root. However, if the constraint conditional expression is reversed with respect to *x*, that is, *low* < *x* instead of *x* > *low*:

```
nsolve(f(x)=Ø,xØ|low<x and x<high →result
```

then *nSolve()* may return a result that is a root, but is not bound by the constraints. As an example consider the function

$$f(x) = x^3 - 6 \cdot x^2 + 11 \cdot x - 6 + \sin(x)$$

with roots

x1 = 0.7137 9958 4872 36

x2 = 1.1085 5630 2169 9
x3 = 2.1404 2730 947

and we want to find *x3*. Various permutations of the constraints give these results:

| | |
|---|---|
| x>2 and x<2.2 | x3 returned |
| 2<x and x<2.2 | x3 returned |
| x>low and x<high | x3 returned |
| low<x and x<high | *x1 returned - incorrect!* |
| low<x and high>x | *x1 returned - incorrect!* |
| x>low and high>x | x3 returned |

If the constraints are numbers, then the conditional ordering is immaterial; *nSolve()* returned the correct bounded root. However, if the constraints are local variables, then we must order the conditional expression such that the independent variable occurs *first*.

*solve()* gives similar interesting results:

| | |
|---|---|
| x>2 and x<2.2 | x3 returned |
| 2<x and x<2.2 | x3 returned |
| x>low and x<high | x3 returned |
| low<x and x<high | undef<0 and x = x2 or undef<0 and x = x1 or x = x3<br>(long solution time, also warning *More solutions may exist*) |
| low<x and high>x | undef<0 and x = x2 or undef<0 and x = x1 or x = x3<br>(even longer solution time, and warning *More solutions may exist*) |
| x>low and high>x | x3 returned |

*Indpendent variable local declaration*

In what may be a related situation, *nSolve()* results seem to depend on whether or not the independent variable is declared local. I will use this program as an example:

```
solvebug(vf)
Func
©Demonstrate nSolve() failure with constraints
©29decØ1/dburkett@infinet.com

Local vfs,tapx,t

© Find estimate tapx for solution
ln(1ØØØ*(vf-.Ø15))→vfs
polyeval({3.618919682914,⁻31.ØØ383Ø334444,76.7374729786Ø3,⁻68.2372Ø1523917,262.4613974175
1,84.916629306139},vfs)+polyeval({⁻3.9287348339733ᴇ⁻7,5.9179552Ø41553ᴇ⁻5,⁻Ø.ØØ3689615561Ø
467,Ø.123Ø899Ø642Ø18,⁻2.756Ø332337Ø98,Ø},1/vfs)→tapx

© Find solution, with bounds from tapx
nSolve(vfts12Ø(t)=vf,t)|t≥tapx-.4 and t≤min({tapx+.4,7Ø5.47})

EndFunc
```

and *vfts120()* looks like this:

```
vfts12Ø(ts)
func

© This expression is all one program line.
polyeval({6.1771Ø2ØØ5557ᴇ⁻14,⁻3.724921779392ᴇ⁻11,9.3224938547231ᴇ⁻9,⁻1.2394592274Ø7ᴇ⁻6,
9.2348545475962ᴇ⁻5,⁻.0036542400520554,.075999519225692},ts)

Endfunc
```

While it may appear complicated, the logic is quite simple.  We want to solve the function *vfts120()* for the input argument *vf*. To speed the solution, we first find an approximate solution for *t*, called *tapx*. *tapx* is in turn the center of a desired solution range *tapx* - 0.4 to *tapx* + 0.4. Finally, we don't want to search for solutions beyond t = 705.47, so that constraint is included, too. As a test case, we try vfts120(100.17) = 0.01613053407. In this case the contraints are about 99.789 and 100.589, so the desired root of 100.17 is included in the bounds. As shown above, *solvebug()* returns about 63.673, which is a root, but outside of the constraints. Note that the constraint conditional expressions are ordered correctly: the independent variable appears first.

If we remove *t* from the *Local* list, then the correct root of 100.17 is returned, but a *Questionable accuracy* warning is shown. Regardless of whether or not *t* is declared local, the correct root is quickly returned without the warning message, if we use *tapx* as an initial guess for *t*, like this:

```
    nSolve(vfts120(t)=vf,t=tapx)|t≥tapx-.4 and t≤min({tapx+.4,705.47})
```

So, it appears that the optimum fix for this problem is to declare the independent variable as *Local*, and also use an initial guess for the independent variable.

# 12.0 C Programming Tips

## [12.1] Advantages of TIGCC over TI SDK

There are two options for C programming on the calculators: TI's proprietary SDK, and the open-source TIGCC. TIGCC cannot be used to develop flash applications, but it has many other advantages over the SDK:

- Global and static variables are allowed in RAM programs with TIGCC.
- Programs can be automatically compressed with TIGCC. This also allows RAM programs of up to 64 KB of uncompressed size rather than just 24 KB.
- TIGCC is given with a static library called TIGCCLIB which contains functions like most ANSI stdio.h and stdlib.h functions, grayscale and sprite routines and many more.
- By default, programs written with TIGCC also run on AMS versions lower than 2.04 without having to install a memory resident emulation program. As an alternative, starting from TIGCC 0.94, the AMS 2.04/2.05 only ROM_CALL method used in the SDK is also supported. The SDK only supports the method incompatible with AMS 2.03 or lower.
- By default, programs written with TIGCC also work on AMS 1, while SDK programs often fail to work even with an emulation program because they use ROM_CALLs available on AMS 2 only. SDK programs even sometimes use ROM_CALLs only introduced in AMS 2.04 or 2.05. Starting from TIGCC 0.94, TIGCC now optionally allows the programmer to use those functions if they wish to abandon AMS 1 compatibility, but does not force him to do so.
- TIGCC is a port of GCC 3, so you get GCC specific features such as GNU C extensions with it.
- Starting from TIGCC 0.94, TIGCC supports passing arguments to functions by registers rather than on the stack as an option. This can create smaller and faster code and make it easier to interface with existing assembly code.
- TIGCC supports third-party static libraries, and a few of those have already been written, such as ExtGraph by Thomas Nussbaumer.
- TIGCC also allows you to program in assembly (not just inline assembly), and you even have the choice between 2 different assemblers.
- TIGCC also allows you to write kernel-based programs, including dynamic libraries and programs which use them (even though I do not recommend to use this feature).

*(Credit to Kevin Kofler)*

## [12.2] Use C function in expressions with AMS 2.05

AMS 2.05 does not allow using C programs in expressions, as TI Basic functions can be used. If *cprgm()* is a C program, this won't work:

```
cprgm(5)+1 → var
```

This is a severe, unnecessary limitation, and TI would be wise to fix it. This seems unlikely since it still exists. However, there is a user hack called *IPR*, which fixes the problem. Get it, with documentation, at *http://sirryl.multimania.com/beta/ipr.zip*

But it gets better! This hack will not work on a HW2 calculator, unless you also install Kevin Koffler's *h220xtsr* hack, from here:

*http://members.chello.at/gerhard.kofler/kevin/ti89prog.htm#h220xtsr*

Many users have successfully used these programs with no problems, but there is no guarantee that they will work in all cases.  Consider yourself warned.

### [12.3]  TI Basic extension app template

The C code below is an example of creating TI Basic extensions in an SDK flash application. The items in angle brackets <> should be replaced with appropriate values for the application. You will need to add more code if there are more than two extensions.

```
///////////////////////////////////////////////////////////////////////
/*********************************************************************/
/*                  TI-Basic Extension App Template                */
/*********************************************************************/
/* by Author (FlashAppAuthor@domain.com)                            */
/*    http://url                                                     */
/* Version 1.0                                                      */
/*********************************************************************/
///////////////////////////////////////////////////////////////////////

/* standard include file*/
#include "tiams.h"

/* Handy equates - "magic numbers" are bad */
/* Alphabetical order highly recommended   */
#define NUMBER_OF_EXTENSIONS <2>
#define FRAME_ENTRIES_PER_EXTENSION 2
#define <EXTENSION1>_OFFSET 0
#define <EXTENSION2>_OFFSET 1
#define HELP_OFFSET NUMBER_OF_EXTENSIONS
#define CONSTANT_FRAME_LENGTH 6
#define FRAME_LENGTH
(CONSTANT_FRAME_LENGTH+(FRAME_ENTRIES_PER_EXTENSION*NUMBER_OF_EXTENSIONS))

/* Function prototypes here */
void <extensionroutine1>(void);                         //This function will be
called when you evaluate MyApp.MyExt1
void subroutine1(EStackIndex, float, float*, float);    //Example of subroutines
float subroutine2(float);                               //Example of subroutines
void <extensionroutine2>(void);                         //This function will be
called when you evaluate MyApp.MyExt2

/* Extension table - these must be in ASCII (usually, this means 'alphabetical')
order */
APP_EXTENSION const extensions[] =
 {
    /* function name #,                    help string #,
                 function index */
    {OO_APPSTRING+<EXTENSION1>_OFFSET,
OO_APPSTRING+HELP_OFFSET+<EXTENSION1>_OFFSET,           <EXTENSION1>_OFFSET},
    {OO_APPSTRING+<EXTENSION2>_OFFSET,
OO_APPSTRING+HELP_OFFSET+<EXTENSION2>_OFFSET,           <EXTENSION2>_OFFSET}
 };
```

```
/* Recommended that you maintain the same order as above */
APP_EXT_ENTRY const extEntries[] =
 {
    {<extensionroutine1>,     APP_EXT_FUNCTION},
    {<extensionroutine2>,     APP_EXT_FUNCTION}
 };

FRAME(tibasicextFrame, OO_SYSTEM_FRAME, 0, OO_APP_FLAGS, FRAME_LENGTH)
    ATTR(OO_APP_FLAGS, APP_NONE)
    ATTR(OO_APP_NAME, "My Flash App")
    ATTR(OO_APP_TOK_NAME, "MyApp")
    ATTR(OO_APP_EXT_COUNT, NUMBER_OF_EXTENSIONS)        /* export extension
functions        */
    ATTR(OO_APP_EXTENSIONS, extensions)                        /* address of
extensions table */
    ATTR(OO_APP_EXT_ENTRIES, extEntries)                /* address of ext entries
table     */
    ATTR(OO_APPSTRING+<EXTENSION1>_OFFSET, "MyExt1")    /* TIOS function names
          */
    ATTR(OO_APPSTRING+<EXTENSION2>_OFFSET, "MyExt2")    /* TIOS function names
          */
                                                               /* Catalog
description follows:    */
    ATTR(OO_APPSTRING+HELP_OFFSET+<EXTENSION1>_OFFSET, "Help for MyApp.MyExt1")
    ATTR(OO_APPSTRING+HELP_OFFSET+<EXTENSION2>_OFFSET, "Help for MyApp.MyExt2")
ENDFRAME

/* Frame pointer required as the first static value in app */
pFrame TIBasicExtensionFrame = (pFrame)&tibasicextFrame;

//Some defines that might be useful if you are used to TIGCC
#define GetArgType(p) (*(unsigned char*)(p))
#define malloc(sz)       HeapDeref(HeapAllocHigh(sz))
#define calloc(NoOfItems,SizeOfItems)  malloc(NoOfItems*SizeOfItems)
#define HeapFreePtr(vr) HeapFree(HeapPtrToHandle(vr))

//This is how you use ROM calls directly.
//The number (291) after "tiamsapi_" is the ROM call number in decimal.
//For some ROM calls that don't have corresponding functions in the SDK,
//you can find the ROM call number (in hex) and the parameters from the
//TIGCC documentation.
void tiamsapi_291(BN*, const BN*, const BN*, const BN*);
#define BN_powerMod tiamsapi_291

void <extensionroutine1>(void)
{
    void subroutine1(EStackIndex, float, float*, float);
    EStackIndex argptr;
    Access_AMS_Global_Variables;
    argptr = top_estack;
    ...
    push_Float(res);    //Can push other types too; this is just an example.
    return;
}

void subroutine1(EStackIndex ptr, float x, float* y, float z)
{
    float subroutine2(float);
    ...
    return;      //Returns result through pointer
}

float subroutine2(float x)
{
    ...
    return result;
}

void <extensionroutine2>(void)
{
    ...
}
```

*(Credit to Bhuvanesh Bhatt)*

# Appendix A: Anti-tips - things that can't be done

This section lists some things that you may want to do, or might think you should be able to do, but the TI-89 / TI-92 Plus or TI Basic do not support. This section might save you some time in finding out that you can't implement a particular feature.

1. Your programs cannot, in general, use the text editor or matrix editor - but see tip [7.10].
2. You cannot write to a text variable.
3. Many built-in functions, for example, *nSolve()* and *NewPlot*, won't accept local variables as arguments. Use global variables, and delete them with *DelVar* when your program exits.
4. The TI-89 / TI-92 Plus do not have built-in functions to create cascading drop-down menus, as used, for example, on the home screen toolbar items. However, the program *MnuPak* (http://www.calc.org/programs/proginfo.php?id=2716) can be used to do this.
5. Variables cannot be larger than about 64K. Trying to create or use a variable larger then this will cause a memory error, and subsequent memory leakage, in which the available RAM mysteriously decreases. The only known fix is a reset.
6. A program cannot archive itself. However, one program can archive and unarchive another.
7. Functions can read from global variables, but cannot store to (write to) global variables. Programs can read and write to global variables. Functions *can* write to their argument variables, which are considered local variables.
8. Functions cannot write to the program I/O display.
9. Programs cannot, in general, return results to the history area, but see tip [7.8].
10. You cannot reference single-row or single-column matrices (vectors) with a single index. Example: *matrix[3]* won't work. This works: *matrix[3,1]*.
11. You cannot use just *ans()* in a function. The TI Basic tokenizer will replace *ans(1)* with the actual value of *ans(1)* the first time the program runs. Instead, use *expr("ans(1)")*. See tip [9.10] for an example. This behavior can also be avoided by archiving the function before running it. *entry()* exhibits the same behavior.
12. You cannot delete statistics plot definitions within a program or function. There is no command to do this. However, see tip [4.15] for a possible work-around.
13. You cannot directly write to data variable elements. See tip [3.8] for a workaround.
14. Number base conversions (▸Hex, ▸Dec, ▸Bin) only work in Auto or Exact modes. In Approximate modes, the base conversions return a *Domain error* message. Keep this in mind if you write programs that use the number base conversions.
15. You must use global variables, not local variables, to do symbolic math in programs. This is in the *89/92+ Guidebook*, page 291.
16. You cannot archive variables beginning with the underscore character '_'. The operating system considers these to be system variables and prevents the archive. The underscore character is typically used to define custom user units.
17. You cannot pass Data variables as program or function arguments. If this is necessary, convert the data variable to a matrix (if all the columns have the same number of rows), and pass the matrix. See tip [3.29] for details.
18. You cannot use elements of lists or matrices as arguments in the *Request* command. To accomplish this, first store the element to a variable, use the variable in *Request*, then store the variable to the element.

# Appendix B: More resources

## [B.1] FAQs (Frequently asked questions)

### *Ray Kremer's Graphing Calculator FAQ*

This is Ray Kremer's ultimate TI calculator FAQ:

http://tifaq.calc.org/

Texas Instruments maintains four FAQs for the TI-89, TI-92 and TI-92 Plus calculators, and the GraphLink. Note that the TI-92 FAQ has a lot more information that either the TI-89 or the TI-92 Plus FAQ, and almost all of it applies to all three calculators.

### *Texas Instruments' FAQ for the TI-89*

- Why did TI produce the TI-89?
- How much does the TI-89 cost?
- Where can I buy a TI-89?
- Will there be a TI-89 Plus?
- Is the TI-89 approved for College Board tests?
- What is the size of the TI-89?
- Is the TI-89 available through the Workshop Loan Program?
- Is the TI-89 compatible with the TI-92 and TI-92 Plus?
- Which ViewScreen LCD does the TI-89 use?
- Does the TI-89 work with the TI-GRAPH LINK?
- What Data Collection tools is the TI-89 compatible with?
- How much memory does the TI-89 have?
- What do Flash and electronically upgradable mean?
- What is Advanced Mathematics Software?
- What is the TI-89?
- What is the difference between Hardware Version 1.0 and 2.0?

***Why did TI produce the TI-89?***
TI listened to educators and students and heard the need for a portable symbolic calculator with more advanced mathematics and engineering features.

***How much does the TI-89 cost?***
The TI-89 is priced at approximately $150 in the U.S.

***Where can I buy a TI-89?***
The TI-89 is available in the U.S. and Canada in limited quantities at some retail stores, Instructional Dealers, and college bookstores.

Some locations of these stores may not carry the TI-89. Please contact your local store to ensure availability.

***Will there be a TI-89 Plus?***
The TI-89 already comes with the Flash memory. It is different from the TI-83 and TI-92, which were introduced before the Flash memory became available on the calculator. Therefore there is no need to introduce a Plus version of the TI-89 with the Flash memory update. TI has four graphing calculator models with Flash capability: TI-73, TI-83 Plus, TI-89, and TI-92 Plus.

***Is the TI-89 approved for College Board tests?***
The TI-89 has been approved for some College Board tests, including AP Calculus and SAT.

***What is the size of the TI-89?***
The TI-89 will be the same size as the TI-83/86: 1.0 x 3.5 x 7.3 (in.).

***Is the TI-89 available through the Workshop Loan Program?***
Yes, teachers can borrow individual TI-89s for short term loans through TI's Workshop Loan Program. Loans are based on availability.

***Is the TI-89 compatible with the TI-92 and TI-92 Plus?***
Yes, the programming functions and data on the TI-89 will transfer directly to and from a TI-92 Plus using the included unit-to-unit link cable. Data and programs containing original TI-92 functions will transfer directly to and from a TI-89.

***Which ViewScreen LCD does the TI-89 use?***
The TI-89 ViewScreen calculators use the same ViewScreen panels as the TI-92.

***Does the TI-89 work with the TI-GRAPH LINK?***
Yes. The TI-89 will works with the TI-GRAPH LINK cables.

***What Data Collection tools is the TI-89 compatible with?***
The TI-89 is compatible with CBL, CBL2, and the CBR.

***How much memory does the TI-89 have?***
The TI-89 provides two types of memory: RAM and user data archive memory. The user available RAM is about 188K and can be used for computations and storing programs and data. The user data archive has 384K of available space for storing additional programs and data.

***What do Flash and electronically upgradable mean?***
Flash is a technology that allows the calculators software to be upgraded electronically. Using the TI-GRAPH LINK cable and the link I/O port, the software can be updated with the latest maintenance updates or new software versions.

***What is Advanced Mathematics Software?***
Advanced Mathematics Software is the name of the software features available on the TI-89. The software is named to identify which version of software is running once electronic updates are available. This is the same Advanced Math Software of the TI-92 Plus, without the Geometry application.

The key features of Advanced Mathematics Software are the Computer Algebra System, differential equations, linear algebra, unit conversions, systems of equations, and 3D graphing with contour plots and rotation.

***What is the TI-89?***
The TI-89 is a vertical graphing calculator that includes a symbolic Computer Algebra System (CAS), Advanced Mathematics Software, and Flash technology that enables electronic upgradability.

***What is the difference between Hardware Version 1.0 and 2.0?***

TI frequently makes running changes to the hardware of its calculator products. If you purchased a TI-89 after July 1999, it may include some minor hardware updates as described below. You can identify these calculators by the statement "Hardware Version 2.0" on the About screen (press F1, A:About...). All TI-92 Plus calculators have Hardware Version 2.0.

- Altered the management of the user portion of the Flash ROM. You will recognize this update only when the Advanced Mathematics Software Version 2.0 operating system is installed. With hardware version 2.0 and the 2.0 operating system, you have more user archive space while the total amount of Flash ROM is unchanged.
- Once a user has loaded the 2.0 operating system and a few applications, the hardware version 2.0 and the original TI-89 are practically the same. This scenario is expected for most users in the future as many great application ideas are floating around.
- Slightly increased the processor speed from about 10 MHz to about 12 MHz.
- TI-92 Plus only: Removed the TI-92 Plus Module compartment.

Note: New TI-89 operating system software, Advanced Mathematics Software v1.05, was required to support the hardware changes. Therefore, Hardware Version 2.0 will reject earlier versions of the operating system software.

***Texas Instruments FAQ for the original TI-92***

http://education.ti.com/product/tech/92/faqs/faqs.html

This FAQ applies to the original TI-92, not the current TI-92 Plus. However, much of the information is still valid, and it answers more questions than the TI-92 Plus FAQ.

- comDenom() (common denominator) function algorithm.
- Changing the order of steps gives different answers sometimes - why?
- Difference between [diamond] [OFF] and [2nd] [OFF].
- fMin() (symbolic minimum) function algorithm.
- fMax() (symbolic maximum) function algorithm.
- factor() function algorithm.
- I have two TI-92s that simplify the same expression differently.
- limit() function algorithm.
- Similar expressions simplify differently - why?
- Simplifications - why are some so slow?
- Sigma(1/(n^3),n,1,infinity) doesn't return an answer - why?
- Simplification ignores path names for variables - why?
- Scrolling a long answers on the Program IO screen.
- Replacing the cover (snap cover) back on the TI-92.
- Product function (symbolic) algorithm.
- Programming language - is the TI-92's like the 82 and 85?
- Programming language of the TI-92 - is it BASIC?
- propFrac() (proper fraction) function algorithm.
- Superscripted numbers appear on multiple output statements - why?
- Soft Warehouse's TI-92 Program Library.
- Symbolic Manipulation (Computer Algebra) - what is it?
- taylor() (taylor series) function algorithm.
- Some numbers not effected when I change certain mode settings-why?
- Summation function (symbolic) algorithm?

- solve() function - why does it not find all the solutions?
- Solve ignores With ( | ) constraints with trig functions - why?
- Store a value to a matrix element.
- nDeriv() (numeric derivative) algorithm.
- Calculator will not accept a key press - why?
- Can I use the link port for my own applications?
- Adding features to the existing TI-92.
- Anti-derivatives - why can't the TI-92 find them for all expressions?
- Can't see displays because of the classroom lighting.
- arcLen() (arc length) function algorithm.
- Cabri Geometry II features not in the TI-92.
- AUTO modes function - what is it?
- Answer in back of the book is different from the TI-92 - why?
- avgRC() (average rate of change) algorithm
- cSolve() (complex solve) function algorithm
- solve() (symbolic solve) algorithm.
- Control pad - how many directions will it move the cursor?
- cFactor() (complex factor) function algorithm
- Complex numbers sometimes round a component to zero - why?
- Circular Definition Error.
- Construct an ellipse in TI-92 Geometry.
- cZeros() (complex zeros) function algorithm
- Does TI supply a TI-92 ViewScreen and the separate parts?
- Division by zero - how can I create this error?
- Display an angle in DMS format on the Program I/O screen.
- Display - does it scratch?
- Differential equations - why doesn't the TI-92 have them?
- DERIVE - does the TI-92 have ALL of DERIVE's features?
- Define a function from a program prompt.
- fMin() and fMax() are hard to use in programs.
- Equation for the graph on the cover of the TI-92 manual.
- fMin(expresn, x) and fMax(expresn, x) return only x values - why?
- expand() function algorithm
- Error FOLDER when I try to copy a var in VAR-LINK why?
- Equation Solver on the TI-92?
- For loop slower on the TI-92 than the TI-8x products - why?
- I only have 70K bytes for user memory - why?
- Gamma Function
- Implicit differentiation
- How "complete" is the TI-92's symbolic manipulation?
- Geometry figure on TI-92 manual - how to create it.
- infinite sums - why does the TI-92 only compute certain ones?
- Memory requirement to open any application on the TI-92?
- Maximum number of variables?
- Locked variables don't show up in open dialog box choices - why?
- Limit() returns an answer when you expect undefined - why?
- Why does the TI-92 returns 0 for limit(sqt(1-x),x,1) when it should be undefined?
- Integration (symbolic) function algorithm
- nSolve() (numeric solve) function algorithm
- nInt() (numeric integration) algorithm
- Print history area on TI-92
- Number bases other than decimal on the TI-92
- nsolve() - why does it sometimes take so long to find a solution?

- Press ENTER twice in a dialog box to save settings - Why?
- min() function - why does it not work on strings?
- Pictures in TI-92 Toolbars
- Phase Planes on the TI-92
- Verifying a symbolic result
- d() (symbolic differentiate) algorithm
- I can't get my cover (snap cover) off. How do I get it off?
- How is the application-specific 68000 different from a regular 68000?
- Graphing and Geometry screens differ by one pixel each way - why?
- Implied multiplication-x[space](a+b) is not read as multiplication?
- Limiting the glare off the screen in classrooms
- Integration - why is a numeric answer returned when I expected symbolic?
- Menu options are fuzzy - Why?
- tCollect() (trig collect) function algorithm
- tExpand() (trig expand) function algorithm
- Trig functions return unexpected answers in DEG mode - why?
- View the name of the variable currently in any editor
- ViewScreen connector - does every TI-92 have it?
- What is under the little screwed in cover plate under the back case?
- When doesn't work as expected in a data variable - why?
- Why does $4^x$ get simplified into $2^{2x}$?
- Why, when I enter solve (sin(x)=0,x) do I get x=@n1*pi? (TI-92)
- zeros() function algorithm

***comDenom() (common denominator) function algorithm.***
Term by term, comDenom(a/b+c/d) reduces (a*d+b*c)/(b*d) to lowest terms, and repeats this process for each additional term.

***Changing the order of steps gives different answers sometimes - why?***
There are times that changing the order of steps in a problem are valid and do not affect the mathematics. In these cases, the end result can "look" different because the simplification process had different forms of expressions to work on each time.

***Difference between [diamond] [OFF] and [2nd] [OFF].***
Let's say the you are in the graphing screen and want to turn the unit off. If you press [2nd] [OFF] the calculator will reset back to the home screen and then turn off. So, when you turn it back on you go back to the home screen instead of where you were.

If you use the [diamond] [OFF] this emulates APD (Automatic Power Down). This means that when you turn the unit back on, you go back to where you were.

***fMin() (symbolic minimum) function algorithm.***
For fMin(expr,var), if d(expr,var) cannot be computed symbolically, a combination golden-section parabolic interpolation search is performed for one local minimum. (R. Brent, "Algorithms for Minimization without Derivatives", Prentice-Hall, 1973.) Otherwise, fMin(expr,var) is determined by solve(d(expr,var)=0,var), filtered by attempting to determine the signs of higher-order derivatives at these candidates. Surviving candidates are compared with the limits of expr as var approaches inf and -inf; and also with points where expr or its derivative is discontinuous.

***fMax() (symbolic maximum) function algorithm.***
fMax() is similar to fMin(). See fMin() for more information.

***factor() function algorithm.***
The methods include the well-known factorizations of binomials and quadratics, together with methods described by Geddes, Czapor and Labahn, "Algorithms for Computer Algebra", Kluwer Academic Publishers, Boston, 1992. Laguerre's method is used for approximate polynomial factorization. (Press et. al: "Numerical Recipes" Cambridge University Press, 1986.)

factor( returns the rational number factored into primes and a residual having prime factors that exceed 65521.

***I have two TI-92s that simplify the same expression differently.***
Why? With slight modifications that have been made to the TI-92 ROM there have been some very specific examples that do not evaluate the same from version to version. Other problems may evaluate differently or not at all. A change may not have been made in the TI-92 symbolic manipulation code to directly correct this example. It most likely was an indirect result of changes made for other reasons. You will also be able to find examples of problems that will not evaluate on later TI-92's that would on earlier versions. This is the nature of dealing with CAS software.

***limit() function algorithm.***
Limits of indeterminate forms are computed by series expansions, algebraic transformations and repeated use of L-Hopital's rule when all else fails. Limits of determinate forms are determined by substitution.

***Similar expressions simplify differently - why?***
The goals of default simplification are to ensure that critical cancellations occur, without unnecessarily applying drastic transformations that might consume an unacceptable amount of time or exhaust memory. Simplification is necessarily a compromise between these conflicting goals.

Minor appearing differences between expressions can cause dramatic differences in the default simplification path. For example, there is an effort to retain most polynomial factoring, but adding a term to a factored polynomial can cause the polynomial to be expanded. Similarly, there is an effort to avoid common denominators, unless denominators share a common factor, so some fractions combine while others don't. As another example, x orders more main than y, so merely changing the names of variables can affect how much expansion and common denominators occurs.

Also, the default simplification path can depend strongly on the mode settings.

There are so many potentially useful mathematical transformations that it is not feasible to fit them all in the limited amount of program ROM. Consequently, a useful transformation that you desire might not be in the TI-92. For example, x*x^n will not simplify to x^(n+1) because this is not one of the rules built in. Another example is integral(x^(n-1),x) will simplify to x^n/n, but integral(x^(n+1),x) will not simplify.

***Simplifications - why are some so slow?***
Some operations can be inherently slow for rather simple operands. Examples include factoring, simplification of fractional powers, and cancelation of hidden factors between non-numeric numerators and denominators. In some cases, intermediate expressions can become quite large despite modest inputs and final results.

If a problem is taking an unacceptable amount of time:

1. Try using a different mode, such as diamond-enter versus enter.
2. Try using an appropriate alternate function, such as nSolve() versus solve() or nINT() versus integral().

3. Try solving a simpler related problem: For example, substitute appropriate simple numbers for some of the variables, or conversely try substituting variables for some of the messy constants or subexpressions.
4. Use a simpler physical model (such as ignoring friction).
5. Use the taylor() function or other means to replace an expression with a simpler approximation.

***Sigma(1/(n^3),n,1,infinity) doesn't return an answer - why?***
The function sum (1/(n^k), n, 1, infinity) (the sum of 1/ (n raised to the k-th power) as n goes from one to infinity) works when k is an even number, but when k is odd the unit simply returns the expression.

Example:

3->k

sum(1/(n^k),n,1,infinity).

The unit returns (in pretty print),

Reason:

Without the Riemann zeta function, there is not an exact closed form of that answer unless k is even and greater than one. DERIVE and Maple have a built-in Riemann zeta function, but the TI-92 doesn't. However, you can always approximate the result by using a large constant in place of infinity.

***Simplification ignores path names for variables - why?***
Path names are simplified when they reference a defined variable. The contents of the defined variable replace the path name in the expression. For undefined variables, there is not any attempt to determine if the paths are referencing the same variable. For example, with the main folder current, x+main\x ENTER will not simplify to 2x even though they are really the same variable.

***Scrolling a long answers on the Program IO screen.***
You cannot scroll answers on the IO screen. You must display the answer on the Home Screen before you can scroll through it.

***Replacing the cover (snap cover) back on the TI-92.***
The cover only fits on the TI-92 one way. Be sure that the tabs on the top sides of the calculator match up with the inserts in the cover. Also, the printing on the outside of the cover should be right side up when placed on the calculator.

***Product function (symbolic) algorithm.***
Iterated multiplication is used for modest numeric limits. Other products are determined by checking for telescoping products and simple patterns. Limits are used for infinite product limits.

***Programming language - is the TI-92's like the 82 and 85?***
The TI-92 contains a superset of the TI-82 and TI-85 programming language. A number of features were added to specifically address needs from TI-82 and TI-85 users like: memory management, deleting variables, enhanced string features, enhanced graphics features, local variables, user defined functions, and parameter passing with programs and functions.

***Programming language of the TI-92 - is it BASIC?***
No. There are a number of features that are similar to the BASIC programming language, but it is not BASIC.

***propFrac() (proper fraction) function algorithm.***

propFrac() uses polynomial division with remainder to determine the quotient polynomial and the numerator of the resulting fraction.

***Superscripted numbers appear on multiple output statements - why?***

Some number are superscripted on multiple output statements because the "pretty print" is on. In the pretty print mode, the unit will calculate the size in pixels of the rectangle needed to hold this output. The coordinates given start in the upper left most corner of the calculated rectangle needed to hold the output. The second OUTPUT command doesn't know the size of the previous output, so if the coordinates given in OUTPUT #2 aren't specified correctly by the user then they will see overlapping or superscripting on the screen.

There are two ways to get around this. First, a person can calculate his/her coordinates for the OUTPUT command through trial and error. Second, they can create a single string containing the results of the multiple outputs and then perform one OUTPUT command using that one large string.

***Soft Warehouse's TI-92 Program Library.***

This library includes supplementary functions and subroutines that add to the functionality of the TI-92. It is available for free download .

The library is currently divided into the following three TI-GRAPH LINK group files:

UNIT.92G implements automatic units algebra and conversion.

ELEM.92G implements pre-calculus mathematics capabilities such as solution of simultaneous nonlinear equations, general regression, contour plots and plots of implicitly-defined functions.

ADV.92G implements more advanced mathematics capabilities such as symbolic solution of differential equations and vector calculus.

Each group includes a corresponding "about...()" program. It states the purpose of the folder, the copyright and abbreviated free copy provisions, then "installs" the folder by performing some housekeeping and setup tasks.

***Symbolic Manipulation (Computer Algebra) - what is it?***

The terms Symbolic Manipulation or Computer Algebra describe a software feature set that use the mathematical rules of algebra and calculus to simplify expressions full of symbols and undefined variables as well as numeric values. Some examples are:

x+3x-y+4y will simplify to 4x+3y

solve(2x-3=y,x) will return the solution x=(y+3)/2

derivative of 3x^4 with respect to x will simplify to 12x^3

(x^3+x)/2x will simplify to (x^2+1)/2

***taylor() (taylor series) function algorithm.***

taylor(u,x,n,a) is sum(limit(d(u,x,k),x,a)*(x-a)^k/k!,k,0,n), computed iteratively to avoid redundant computation of lower-order derivatives.

***Some numbers not effected when I change certain mode settings-why?***
Check to see what type of numbers you are using. The "Display Digits" and "Exponential Formats" do not effect integers and rational numbers.

***Summation function (symbolic) algorithm?***
Iterated addition is used for modest numeric limits. Other sums are determined from formulas given in references such as "Standard Mathematical Tables and Formulae", CRC Press, together with tests for telescoping sums. Limits or special formulas are used for infinite summation limits.

***solve() function - why does it not find all the solutions?***
The TI-92's solve() function attempts to solve an equation analytically to get an exact representation of the solution(s). When it cannot do so, it uses numerical search techniques to attempt to find approximate solutions. These search techniques cannot guarantee to find any or all solutions. When multiple solutions are found, they cannot be guaranteed to be consecutive ones. Search techniques are highly sensitive to the nature of the equation and the form in which it is given, the properties of the individual functions, the fixed precision limitations of the floating point arithmetic, and any boundaries placed upon the search.

In the case of sin(x)=cos(x) the solve() function can determine the exact solutions and can represent them as an infinite family using the "arbitrary integer" notation (@n1). However, in the case of cos(x)=-2e^(-2x), the equation is transcendental. The number of solutions is infinite, and the solutions do become closer and closer to the roots of cos(x) as x moves toward infinity. However, the solutions are not periodic and, in general, can only be approximated.

A very useful technique when using the solver is to graph the left and right sides of the equation and look for intersections on the graph. Or, graph the difference between the left and right sides and look for crossings of the x-axis. Investigating the equation graphically often suggests reasonable bounds that can be placed on the solve() function using the with (|) operator. Providing reasonable bounds often greatly improves the success of the numerical search techniques.

***Solve ignores With ( | ) constraints with trig functions - why?***
If the TI-92 is able to find an exact solution to an equation, and this solution is an infinite family of solutions, the restrictions after the With operator are ignored. The answer returned will contain the arbitrary integer notation and will not return specific answers within the given range.

If the TI-92 is unable to find an exact solution, then it resolves to using a numeric search for the solutions (except in Exact mode). During a numeric search, the TI-92 will apply the constraints to the solutions returned. If nsolve is used, the solution returned will be within the given range. However, nsolve will only return one solution.

If the TI-92 is able to find a finite many exact answers, then the constraints will also be applied.

Examples:

solve(sin(x/2)=cos(x/2),x) | x>=0 and x<2Pi returns ((4@n -3)*pi) / 2 (radian/ auto mode)

nsolve(sin(x/2)=cos(x/2),x) returns -42.4115 (radian/auto mode)

nsolve(sin(x/2)=cos(x/2),x) | x>=0 and x<2Pi returns 1.5708 (radian/auto mode)

solve(sin(2x)=cos(x),x) returns a list of several values (radian/auto mode)

solve(sin(2x)=cos(x),x) | x>=0 and x<2Pi returns list of answers in given range (radian/auto mode) nsolve(sin(2x)=cos(x),x)  returns -10.9956   (radian/ auto mode)

nsolve(sin(2x)=cos(x),x) | x>=0 and x<2Pi  returns 4.71239   (radian/auto mode)

***Store a value to a matrix element.***
If mat1 is a 3x3 matrix, then 12->mat1[2,3] will store the value of 12 in the 2nd row, 3rd column position.  The TI-8x products used parenthesis instead of square brackets to do this.  Parenthesis on the TI-92 are reserved for function and program calls.

***nDeriv() (numeric derivative) algorithm.***
nDerive(f(x),x,h) => (f(x+h)-f(x-h))/(2h)

***Calculator will not accept a key press - why?***
A key may be stuck down, putting it in the active position. This blocks any other key from being accepted and your TI-92 may appear to be locked up. Unstick the key and you should be ready to operate.

***Can I use the link port for my own applications?***
No, the link port was designed to optimize our communications speed and the protocal is not general enough for other applications.

***Adding features to the existing TI-92.***
Through the use of programs, user-defined functions, and geometry macros, you can add functionality to the TI-92.  User written programs and functions can be downloaded from the program archive

You can also upgrade your TI-92 with the TI-92 Plus Module with Advanced Mathematics Software.

***Anti-derivatives - why can't the TI-92 find them for all expressions?***
Finding anti-derivatives can be very difficult, requiring more program ROM than is available. Moreover, it is theoretically impossible to express all anti-derivatives as a finite composition of the functions and operators built-into the TI-92.   However, the TI-92 will find anti-derivatives for most integrands encountered in most calculus texts and in tables such the the CRC handbook.

There is a function nInt() for finding numerical values of definite integrals where anti-derivatives can not be found using the symbolic techniques.  In Auto mode, numerical techniques are used automatically when needed for definite integrals.

***Can't see displays because of the classroom lighting.***
Try adjusting the display contrast using [diamond] [+] to darken the display or [diamond] [-] to lighten the display.  Also try using the cover as a stand to improve your viewing angle.

***arcLen() (arc length) function algorithm.***
The implemented textbook definition  arclen (u,x,a,b) = integral (sqrt (1 + d(u,x)^2), x, a, b) is actually a more useful arc-DISPLACEMENT:  The integrand is positive for real u.

If the lower limit exceeds the upper limit, then dx is negative, making the result negative.

***Cabri Geometry II features not in the TI-92.***
Conics, circle filling, multiple animation, and construction replay.  Also some computer specific functions (like color selections and cut/copy/paste) have been removed.

Cabri II is not built in the TI-92.  Texas Instruments worked with the authors of Cabri Geometry II to jointly develop a geometry package for the TI-92.  Many of the features of Cabri II are in the TI-92.

***AUTO modes function - what is it?***

The AUTO mode will, through a predetermined method, display answers as either exact or approximated results.  This mode will display default to an exact answer unless the answer becomes "messy" or would be better understood as an approximation.

***Answer in back of the book is different from the TI-92 - why?***

When a symbolic result is different than what you expected you can use several tools in the TI-92 to help verify that the result is equivalent:

1. Try subtracting the result from the expected result to see if simplifies to zero.
2. Try graphing the result you expected and the result you got from the TI-92.
3. Try using some of the manipulation features like expand(), factor(), comDenom(), etc. to see if you can force further simplification of either result or better yet, their difference.
4. Try substituting appropriate random values for some or all of the variables in each of the two expressions.

***avgRC() (average rate of change) algorithm***

avgRC(f(x),x,h) => (f(x+h)-f(x))/h

***cSolve() (complex solve) function algorithm***

cSolve() is the same as solve() , except the domain is temporarily set to complex so that complex solutions are sought and not rejected.

***solve() (symbolic solve) algorithm.***

solve() uses factoring together with the inverses of functions such as ln(), sin(), etc.  Except in exact mode, these techniques might be supplemented by heuristics to isolate solutions, followed by nSolve() to refine approximations.

***Control pad - how many directions will it move the cursor?***

The control pad moves in eight directions.

***cFactor() (complex factor) function algorithm***

cFactor() is the same as factor(), except the domain is temporarily set to complex so that complex factors are sought and not rejected.

***Complex numbers sometimes round a component to zero - why?***

The TI-92 rounds the real or imaginary part to 0.0 if its magnitude is less than 1E-13 times the magnitude of the other component.  This is done to recover real or pure-imaginary results when small roundoff errors make both components non-zero.

***Circular Definition Error.***

A software change was made in the TI-92 to detect function and program calls that have expressions of symbolic variable argument names that are the same as those used to define the function or program. The error "Circular definition" will occur if that is detected.

Unexpected results can occur when a function or program is called with some form of the same variable name that was used when defining the argument in the function or program definition.

For example, on the home screen:

```
define f(x)=ln(ln(x)+x)              (Done)
f(x+1) simplifies to ln(ln(x+1)+x+2)    (unexpected result!)
```

To avoid this, functions and programs should be defined with argument variable names that WILL NOT be used as arguments during subsequent function or program calls. We suggest you avoid choosing variable names like x,y,z,a,b,c,... for argument names when defining a function or program. These common variable names are frequently used and could easily be included as an argument in a function or program call. Instead, choose variable names like xx, xtemp, yy, ytemp, zz, ... for argument names when defining a function or program. By doing this, you can then use x,y,z,a,b,c... when you call the function or program. This minimizes any chance that you will call a function or program with an argument name that could produce an unexpected result.

For example, on the home screen:

```
define f(xx)=ln(ln(xx)+xx)            (Done)
f(x+1) simplifies to ln(ln(x+1)+x+1)  (expected result)
```

One common area where the simplification problem can occur is in the study of composition of functions. For example, simplifying f(g(x)) with defined functions f(x) and g(x) to see the composition of "f composed on g". When the functions f(xx) and g(yy) are defined, the simplification of f(g(x)) will then give expected results.

For example, on the home screen,

```
define f(xx)=xx^2              (Done)
define g(xtemp)=xtemp+1        (Done)
define h(yy)= yy^(1/2)+1       (Done)
f(g(x)) simplifies to (x+1)^2  (expected result)
h(f(x)) simplifies to abs(x)+1 (expected result)
```

If the arguments in a function or program call are not symbolic, then all simplification will work as expected. Composition of functions is also commonly studied when graphing y2(y1(x)). Because the graphing application on the TI-92 is completely numeric, all composition of functions in graphing work as expected. However, the TI-92 graphing functions y1(x),y2(x),..., y99(x) enforce the use of x as the argument variable name. Therefore, if these system graphing functions are used to study composition of functions symbolically on the home screen, unexpected results can occur.

You can determine whether your version of the TI-92 includes the newer software by doing the following:

On the home screen enter, define f(x)=x^2 (done). Then enter f(x+1). If you get an error "Circular definition" then you have the newer software. If you get a simplified result of (x+1)^2, then you have the older software.

***Construct an ellipse in TI-92 Geometry.***

- Construct a circle.
- Construct a point (A) on the circle.  Construct a point (B) inside the circle.
- Connect those points.
- Construct the perpendicular bisector of the segment between points A and B.

From here you have two options to construct the ellipse.

- Construct the locus of the bisector as point A moves on the circle -- the envelope of lines is the ellipse.  This means that the line we constructed is tangent to the ellipse.  The point of tangency is *not* the midpoint of the segment.

- Construct a segment from the center to the point A. Construct a point (C) at the intersection of the bisector line and this radius. Construct the locus of point C as point A moves on the circle. That will form an ellipse.

***cZeros() (complex zeros) function algorithm***
cZeros(expr,var) is expr>list(cSolve(expr=0,var),var).

***Does TI supply a TI-92 ViewScreen and the separate parts?***
Since every TI-92 sold in the US has the ViewScreen remote panel connector TI does not supply the separate VSCalc like other models of graphing calculators. We also expect more demand for the ViewScreen LCD panel for the TI-92 than for previous models because teachers can buy the TI-92 first and then upgrade to the ViewScreen later.

Dealers can manage their inventory better by just stocking the TI-92 and VS-LCD and putting them together as needed to make up ViewScreen packages. TI only sells those two parts to dealers and does not sell a 92VS combination. TI has adjusted the price on the 92VS-LCD so that a TI-92 and 92VS-LCD together have the same dealer price as the ViewScreen was quoted before.

We still expect dealers to list the 92 ViewScreen on their price sheets.

***Division by zero - how can I create this error?***
A division by zero error is only created while using the Graph/Math functions. You will not encounter a division by zero error when working with the CAS on the home screen or in a program.

***Display an angle in DMS format on the Program I/O screen.***
You can only display values in Degrees-Minutes-Seconds format on the Home Screen.

***Display - does it scratch?***
Yes, the display can scratch. Please be careful not to touch the display with pencils, erasers, or any sharp objects.

***Differential equations - why doesn't the TI-92 have them?***
There are several reasons why a product feature is included or excluded. To make products affordable, we have to limit the amount of memory (ROM) used for the system software; more ROM means higher cost. Therefore, we spend a great deal of time talking to educators about the features that are most useful in the classroom to the most students. For example, there are much fewer students taking college differential equations than taking high school algebra.

The features that have been on previous products that are not on the TI-92 are: differential equation graph mode, some extra linear algebra functions (eigenvalues,eigenvectors, LU decomposition, cond), and an interactive solver.

***DERIVE - does the TI-92 have ALL of DERIVE's features?***
The computer algebra software in the TI-92 is not DERIVE, in whole or part. It was jointly developed by Texas Instruments and the authors of DERIVE. It will do some things that DERIVE will not do and DERIVE will do some thing the TI-92 will not do. Generally, however, it is a fully capable symbolic manipulation package.

***Define a function from a program prompt.***
To Define a function from a program prompt, convert the function contents to a string (if it is not already) and use the expr() function: expr("function string"&"store f(x)")

Example:

```
...
inputStr "enter your function in x", var1
expr(var1&"store y1(x)")
...
```

### *fMin() and fMax() are hard to use in programs.*

The functions fMin() and fMax() return a list of candidate "x values" where minimums or maximums could occur. The result is returned in the form of " x=2 or x=4 or x=19 ...". To get at the actual values of the candidates, use the exp>list() conversion function. The exp>list() function returns the right hand sides of the x=2 or x=4 ... (like) boolean results.

### *Equation for the graph on the cover of the TI-92 manual.*

The equation for the graph found on the cover of the TI-92 manual is:

$$z(x,y) = \frac{20\cdot\cos\left(\frac{x^2+y^2}{4}\right)}{x^2+y^2+\pi}$$

WINDOW:

eyetheta = 20
eyephi = 70
xmin = -5
xmax = 5
xgrid = 20

ymin = -5
ymax = 5
ygrid = 20

zmin = -1.2953 (approx)
zmax = 6.3662 (approx)
zscl = 1

### *fMin(expresn, x) and fMax(expresn, x) return only x values - why?*

If you have the x values where minimums or maximums occur, then it is easy to get the expression values at those points by using the "with" operator. If we were to return instead the expression values that are maximum or minimum, then the corresponding x values would be lost.

### *expand() function algorithm*

Polynomial expansion is done by repeated application of the distributive law, interleaved with use of the associative and commutative laws to collect similar terms.

### *Error FOLDER when I try to copy a var in VAR-LINK why?*

The copy var utility is for copying a variable to another folder, retaining the same variable name. If you want to make a copy of a variable in the same folder you must first copy the var to another folder, rename it, and then copy it back to the original folder.

### *Equation Solver on the TI-92?*

The 92 doesn't have a Solver editor like the TI-85, but the TI-92 does have a more general Solve().

For example:

```
     Solve(x^2+3y+4z=26,x)|y=2 and z=4        x=2 or x=-2
```

or symbolically

```
     Solve(x+2y+7z=17,x)        x=-2y-7z+17
```

***For loop slower on the TI-92 than the TI-8x products - why?***
The For..EndFor loop on the TI-92 uses its arguments to compute the remaining step count every time thru the loop. This provides great flexibility. If variables are used to specify the step size and end value, then these variables, along with the index variable, can be altered within the body of the loop, thus affecting how the loop operates.

The TI-8x products did not do this. They allow the index value to be altered within the body of the loop, but the end value and the step value are computed only once when the loop is started.

***I only have 70K bytes for user memory - why?***
The TI-92 system has a lot of integrated software and it needs memory (RAM) to allow all the features to work correctly.

***Gamma Function***
Define gamma(n)=integrate(x^(n-1)*e^(-x),x,0,infinity)

gamma(5) returns 24. If n is an integer, gamma(n) returns (n-1)!

***Implicit differentiation***
Why does d(x^2 + y^2, x) return 2x, not -2x/(2y)? What is happening is that since you are taking the derivative with respect to x, y is viewed as a constant. A constant squared is still a constant and the derivative is zero, hence the answer.

Here is a first derivative implicit differentiating function based on a more general function found in DERIVE.

```
define impdif (uu, xx, yy)=-d(uu,xx)/d(uu,yy)
```

where uu is an expression that equals 0, and d() is the derivative function. The variables xx and yy are used to avoid a circular definition error that will occur when you differentiate a function of x and y.

Then, enter, for example:

```
impdif(x^2+y^2,x,y)
```

-x/y is returned, which is the reduced form of -2x/(2y)

***How "complete" is the TI-92's symbolic manipulation?***
The symbolic manipulation features set was designed with teachers around the country to address the needs of Algebra I, Algebra II, Trig, Pre-Calculus, and Calculus courses.

***Geometry figure on TI-92 manual - how to create it.***
To create the locus on the manual follow these steps:

- Go to the geometry editor.
- Draw a segment. (F2, 6)
- Put a point on the line a couple of centimeters from the left end point. (F2, 2)
- Create a perpendicular line through the line segment at this point. (F4, 1)
- Draw a circle with the center on the perpendicular line and the radius point being the intersection point of the segment and perpendicular line. (F3, 1)
- Measure the distance from the left endpoint to the intersection point of the circle and the segment. (F6, 1)

- Now you need to transfer that measurement onto the circle. Select F4, 9 (measurement transfer). Then select the measurement from step 6. Then select the circle. Next select the intersection point of the circle and the segment. A point will appear on the circle.
- Reflect this point across the perpendicular line. (F5, 4, select the point, then select the perpendicular line) A new point should appear on the other side of the perpendicular line.
- Hide the perpendicular line and the first point created on the circle. (F7, 1, click on the line, click on the point)
- Now change the format of the Locus to be nonlinked points. (F8, 9, set Link Locus Points to OFF)
- The last step is to draw the locus. Press F4, A, select the point reflected across the perpendicular line and then select the intersection point.

The locus will appear after the last step.

***infinite sums - why does the TI-92 only compute certain ones?***
The TI-92 symbolic math pack can only recognize certain sums. We built in what we thought was most needed to support the Calculus curriculum.

***Memory requirement to open any application on the TI-92?***
Geometry requires about 23K bytes of free memory to start a new session. All other applications require less than 100 bytes to create/open an empty file.

Note: The amount of free memory required to open existing files is dependent on the size of the file.

***Maximum number of variables?***
If the data types are small enough that free memory does not become an issue, you can store approximately 800 different variables.

***Locked variables don't show up in open dialog box choices - why?***
The purpose of locked variables was to keep the variable from changing. All of the editors (Program, Geometry, Data/Matrix, Text) edit the variable directly in memory (they don't work on a copy of the variable, but instead the only copy of the variable). So, to edit one of these variable, you must first unlock it from either the Var-Link dialog box or using the Unlock command.

***Limit() returns an answer when you expect undefined - why?***
***Why does the TI-92 returns 0 for limit(sqt(1-x),x,1) when it should be undefined?***
The limit actually does exist from the right side, it is just an imaginary solution not a real one. That is where the problem arises. Right now, the TI-92 does not filter out answers like this. In fact, an answer should only be returned if you approach 1 from the left side. From both sides and from the right side it should not exist. This is simply a limitation of the TI-92 (and some other CAS systems).

In cases like this, it is a good idea to check the problem using two or three different methods.... limit on home screen, graph the argument, and use the table.

***Integration (symbolic) function algorithm***
Antiderivatives are determined by substititutions, integration by parts, and partial-fraction expansion, much as described by J. Moses ("Symbolic Integration: The Stormy Decade", Communications of the ACM, August 1971, Volume 14, Number 8, pp. 548-559.) Antiderivatives are NOT computed by any of the Risch-type algorithms.) Definite integrals are determined by subdividing the interval at detected singularities, then for each interval, computing the difference of the limit of an antiderivative at the upper and lower integration limits. Except in exact mode, nINT() is used were applicable when symbolic methods don't succeed.

***nSolve() (numeric solve) function algorithm***

nSolve() uses a combination of bisection and the secant method, as described in Shampine and Allen "Numerical Computing: An Introduction", W.B. Saunders Co., 1973.

***nInt() (numeric integration) algorithm***

nInt() uses an adaptive 7-15 Gauss-Kronrod quadrature somewhat like the algorithm described in Kahanner, Moler and Nash.

***Print history area on TI-92***

You can save the entries in the home screen to a text file, send this to the computer and print it. To save the history area as a text object, do the following:

- From the Home Screen, press F1.
- Select Save As....
- Give the text file a name.

You can then open this file in the text editor of the TI-92 or send it to the computer and print.

***Number bases other than decimal on the TI-92***

The TI-92 cannot work with numbers in bases other than decimal without a special program. Check out the Soft Warehouse TI-92 Program Library for such a program.

The reason the TI-92 does not have this function is simply that the teachers we spoke with when designing it thought that other functions were more important than base conversions/arithmetic.

***nsolve() - why does it sometimes take so long to find a solution?***

The nSolve() function uses symbolic techniques to simplify the equation before applying the search technique. Depending on the equation, this simplification can take significant time. For example, solving the Time-Value-of-Money equation for the interest rate can cause expansion of a high-degree polynomial.

func-type User-defined functions are not simplified prior to the numeric search. Consequently, to avoid the time-consuming simplification, use the program editor to define your equation in the following form:

```
:myeqn ()
:Func
:   ... = ...
:EndFunc
```

Then on the home screen enter an expression such as:

```
nSolve (myeqn(), x)
```

Also, using the "with" operator to appropriately constrain the search range can greatly speed the solution. For example, it is very important to restrict the interest rate to positive values when solving the Time Value of Money equation for the interest rate, and it is also helpful to restrict it to be less than some generously large number for a realistic interest rate.

***Press ENTER twice in a dialog box to save settings - Why?***

Dialog boxes can contain several user interface tools: text messages, drop down menus, and input fields. Many dialog boxes require choices or input from the user on several lines before closing down the box. We use the ENTER key to close down the box. So, for both drop down menus and input fields there is a 2 level use of the ENTER key. When inside a drop down menu, the ENTER key can be used to make a selection in that menu without closing down the box. In an input field, ENTER has 2 levels of use also. When your typing is finished, ENTER will simply high light the text without closing down the box. The second ENTER press from either a drop down menu or an input field will close down the entire dialog box.

***min() function - why does it not work on strings?***

The min() function was designed to work on two numeric expressions, two lists, two matrices, or a single list or matrix.

```
define minstrng (str1, str2) = when (str1 < str2, str1, str2)
```

defines a function that returns the alphabetically lesser of two strings.

***Pictures in TI-92 Toolbars***

You can use a picture as a Toolbar title only if the picture has dimensions of 16 x16 pixels.

Use "StoPic picVar [, pxlRow, pxlCol] [, width, height]" to define the picture.

***Phase Planes on the TI-92***

The 92 can do phase planes in sequence graphing mode. It is a sequence plot where you show one function vs. another function.

Differential equation phase planes are not possible on the TI-92.

***Verifying a symbolic result***

When a symbolic result is different than what you expected you can use several tools in the TI-92 to help verify that the result is equivalent:

- Try subtracting the result from the expected result to see if simplifies to zero.
- Try graphing the result you expected and the result you got from the TI-92.
- Try using some of the manipulation features like expand(), factor(), comDenom(), etc. to see if you can force further simplification of either result or better yet, their difference.
- Try substituting appropriate random values for some or all of the variables in each of the two expressions.

***d() (symbolic differentiate) algorithm***

d() repeatedly uses the chain rule together with the well-known formulas for the derivatives of sums, products, sinusoids, etc.

***I can't get my cover (snap cover) off. How do I get it off?***

Pull back on one corner of the cover, "burp it like Tupperware" and the cover should come off.

***How is the application-specific 68000 different from a regular 68000?***

It's a 68000 with:

- Low-power static capability.
- The application specific is our logic we tie around the 68000 to allow us to have our choice features like low battery detect, extra avoid memory-loss circuitry, talk to our LCD, keyboard, etc...

***Graphing and Geometry screens differ by one pixel each way - why?***
In function graphing, it is very important to teachers that the origin pixel be at (0,0) when abs(xmin) = xmax (e.g., -10 to 10). So, we had to make the screen an odd number of pixels high and wide. This was a change made from the TI-81 to the TI-82 and has stuck on subsequent products.

On the Geometry screen, it is not as important that the default window be symmetrical since the user changes the window interactively, instead of on a menu. We shifted the origin one pixel to have it at (0,0) and simply did not delete the rightmost column of pixels. The default window is from -4.148 to +4.103 in the x direction.

***Implied multiplication-x[space](a+b) is not read as multiplication?***
The only significance that the parser gives to white space is that it terminates tokens. Thus, abcd is a single name, but ab cd are two separate names, because the space caused the name parsing algorithm to stop after the b.

*Spaces do not directly create implied multiplication.* Implied multiplication occurs when two valid factors are found in succession without any other binary operator between them. When the next non-blank following a valid name is a left parenthesis, the name is assumed to be a function/program reference.

***Limiting the glare off the screen in classrooms***
Try adjusting the display contrast using [diamond] [+] to darken the display or [diamond] [-] to lighten the display.

Hold the unit at a different angle to change what is being reflected in the display.

We used this type of display to increase the the brightness of the pixels in the display.

***Integration - why is a numeric answer returned when I expected symbolic?***
If you entered an integration problem and received a numeric answer when you expected a symbolic one, there are a couple of reasons.

First, see if the variable you are integrating with respect to has a value stored to it (i.e. X has a value of 5). If it does, you will need to delete that variable or use a different one. You can delete the variable by pressing F6, Enter from the home screen or by using the DelVar command.

If the variable did not have a value stored to it, the TI-92 may have resorted to a numerical approach to find the solution for the problem. This could happen when the problem you are solving is too 'complex' for the symbolic algorithm to solve.

***Menu options are fuzzy - Why?***
Some of the commands are not accessible from all screens or modes. If a command appears unreadable, it means that particular command line is unaccessable from the current mode. It will be available from a different screen or application.

***tCollect() (trig collect) function algorithm***
tCollect() repeatedly uses the well-known formulas for products and integer powers of sines and cosines, such as those given in the CRC tables.

***tExpand() (trig expand) function algorithm***
tExpand() repeatedly uses the well-known formulas for the sine and cosine of multiple angles, angle sums and angle differences, such as those given in the CRC tables.

### *Trig functions return unexpected answers in DEG mode - why?*

Internally, the trig functions always convert to radian measure.  DEG (degree) mode simply multiplies trigonometric arguments by pi/180 before proceeding with simplification.

Thus depending on the arithmetic mode (EXACT, APPROX or AUTO) and whether or not the trigonometric argument contains floating-point constants, you might see pi/180 in a result.

### *View the name of the variable currently in any editor*

You can view the name of a variable currently in an editor of the TI-92.  From inside any of the editors (Text,Program,Data/matrix,Geometry) go to the Save As dialog box (under the F1tools pull down menu) and see the name of the variable in the title of the dialog box.

### *ViewScreen connector - does every TI-92 have it?*

All TI-92s sold in the United States** will have the ViewScreen connector.

We have heard from teachers for a very long time how nice it would be to have each calculator drive a ViewScreen.  Since the TI-92 ViewScreen electrical system is designed into every TI-92 unit, we only had to add the connector to make each TI-92 ViewScreen capable.

This will allow students to easily show something to the class on the teacher's remote display panel. It will also allow teachers to purchase a TI-92 for evaluation and easily upgrade to the ViewScreen capability later.

**TI-92s sold in Europe come in two versions - the regular TI-92 and the TI-92 ViewScreen compatible calculator.

### *What is under the little screwed in cover plate under the back case?*

This is where the back up battery is stored.

### *When doesn't work as expected in a data variable - why?*

The when comparison doesn't work as some people would expect.  If you are trying to test each element in a column to determine which element is entered in another column, you will need to use the following:

```
seq(when(c1[e]>X, 1,2,3),e,1,dim[c1])
```

You have to 'force' the when to compare each element by placing it within a sequence.

### *Why does $4^x$ get simplified into $2^{2x}$?*

Bases are factored into primes to help recognize cancellations. For example,

$$\frac{6^x}{4^x} \Rightarrow \frac{2^x 3^x}{2^{2x}} \Rightarrow \frac{3^x}{2^{2x-x}} \Rightarrow \frac{3^x}{2^x} \Rightarrow \left(\frac{3}{2}\right)^x$$

### *Why, when I enter solve (sin(x)=0,x) do I get x=@n1*pi? (TI-92)*

This equation has an infinite number of points where sin(x)=0.  For example, sin(pi)=0, sin(2*pi)=0, sin(3*pi)=0 and so on...

The TI-92 is showing that this equation has this infinite set of solutions by displaying @n1*pi.  The integer suffix of "@n" increases for successive problems, then wraps back to 1 after 255.

Think of "@n" as denoting "@rbitrary iNteger".

### *zeros() function algorithm*

zeros(expr,var) is expr>list(solve(expr=0,var),var)

### *Texas Instruments FAQ for the TI-92 Plus*

*http://education.ti.com/product/tech/92p/faqs/faqs.html*

- What types of 1st- and 2nd-order Ordinary Differential Equations will the TI-92 Plus solve symbolically?
- What types of Systems of Equations can be solved using the TI-92 Plus?
- What is Advanced Mathematics Software?
- How much memory does the TI-92 Plus have?
- Are there any engineering applications for the TI-92 Plus?
- Is user data archive memory erased by reset?
- Are the TI-92 and TI-92 Plus compatible? What CANNOT be shared between the TI-92 and the TI-92 Plus?
- Is the TI-92 Plus allowed on College Board tests (SAT, ACT)?
- Are there any support materials or workbooks for the TI-92 Plus?
- What are the appropriate courses for the TI-92 Plus?
- How much do the AMS upgrades for the TI-92 Plus cost?
- Is Texas Instruments still selling the TI-92 Plus Module?
- Does the TI-92 Plus use the same ViewScreen as the TI-92?
- What Data Collection tools is the TI-92 Plus compatible with?
- What are the main differences between the TI-92 and TI-92 Plus?
- How much does the TI-92 Plus cost?
- What do Flash and electronically upgradable mean?
- Does the TI-92 Plus work with the TI-Graph Link?

### *What types of 1st- and 2nd-order Ordinary Differential Equations will the TI-92 Plus solve symbolically?*

1st Order: separable, linear, Bernoulli, y'=f(a*x+b*y), exact, integrating factor independent of x, integrating factor independent of y, and homogeneous.

2nd Order: linear constant coefficient (and a few variable coefficient), missing y, missing y', missing x, missing both y' and x.

### *What types of Systems of Equations can be solved using the TI-92 Plus?*

solve(...) and cSolve(...) return real or complex solutions of linear or nonlinear systems of equations. For linear or polynomial systems, they use the Gaussian or Groebner elimination method to seek all solutions. Otherwise, they use a damped-Newton/steepest-descent method to seek an approximate solution close to the user's guess, which defaults to 0.0 for each component.

### *What is Advanced Mathematics Software?*

Advanced Mathematics Software is the name of the software features available on the TI-92 Plus. Features of Advanced Mathematics Software are:

- Symbolic Differential Equation Solving
- Numeric Differential Equation Graphing
- Additional Linear Algebra, e.g. eigenvalues, eigenvectors, functions of matrices, LU and QR decompositions

- 3D Rotation and Contour Plots
- Systems of Equations Solving
- Units of Measure
- Interactive Numeric Solver
- Expanded Statistics features, like Sine and Logistic Regression models, and a Modified Box Plot
- Geometry Enhancements - Check Properties for Member and Equidistant, and expanded Redefinition tool
- Languages in Geometry - menus and messages in English, Spanish, Italian, French and German.
- Assembly language programming
- Hexadecimal and binary operations and conversions
- Function definition and evaluation with the same variable, and function compositions. (Circular Definition Error issue)
- BldData command to make graphical data available as a data variable (multiple lists) for statistical modeling or further analysis.
- isPrime function for finding prime numbers, and improved integer factoring
- Improvements to Polar/Degree mode for complex numbers
- Resizing matrices from the Data/Matrix Editor
- Implicit Plots via the 3D grapher

***How much memory does the TI-92 Plus have?***
The TI-92 Plus provides two types of additional memory: RAM and user data archive memory. The user available RAM is about 188K and can be used for computations and storing programs and data. The user data archive has 384K of available space for storing additional programs and data.

***Are there any engineering applications for the TI-92 Plus?***
Yes. Currently there are three Flash Applications regarding engineering:

1). EE200
Designed primarily for 1st-year and 2nd-year college students in circuit analysis coursework, EE200 is an educational tool. It requires students to know in advance relationships between variables, units appropriate to a particular problem and formulae to express units and their relationships. In using the many advantages of EE200, students can:

- Avoid math and units errors
- View the units appropriate to inputs and solutions
- Convert automatically between units
- Augment the TI calculators' 125 built-in units in 27 categories with 145 more units and 25 additional
- Unit-categories relevant to Electrical Engineering
- Save formulae and answers from homework or exam questions to check work later
- Use ordinary language and notation in developing formulae and units to solve problems
- Be alerted in advanced to incorrect, unbalanced formulae and units
- Import equation sets from da Vinci Technologies' EE-Pro

And more

2). ME*Pro
Helps solve a comprehensive set of equations used by mechanical engineering students and professionals. With a power browser engine and unit manager, it covers over 1000 equations (with 100 pictures) covered in 12 subjects such as Heat Transfer, Strength of Materials, Machine Design, Refrigeration, Pumps, Waves and Oscillations, Fluid Mechanics and Gas Laws. 57 analytical routines

solve topics such as Steam Tables, Thermocouples, Section Properties and Efflux. An interactive reference table provides data tables for refrigerants, water properties, Laplace and Fourier Transforms, and fuels and combustion properties.

3). EE*Pro
Is organized into three convenient sections of Analysis, Equations and Reference useful to solve problems encountered by students and professionals of Electrical Engineering. The user interface is menu driven showing the basic organization into topics and sub-topics. The equation section has over 700 equations while the reference section gives access to data commonly needed by practicing professionals.

***Is user data archive memory erased by reset?***
Reset affects the user data archive memory just like it does RAM.  All data stored in the TI-92 Plus will be erased when you reset the unit (with either the Memory, F1, Reset or the Lock + 2nd + On key sequences).

***Are the TI-92 and TI-92 Plus compatible?***
***What CANNOT be shared between the TI-92 and the TI-92 Plus?***
If original TI-92-specific files are created on the TI-92 Plus and sent to a TI-92, they should run in many cases (including PIC files, text files, strings and Geometry Figures and Macros). Exceptions could be user-defined programs (especially symbolic programs) and functions, GDB's (graph databases), sysdata, and other system variables, and data variable datatypes. Also, lists, matrices and data variables that are strictly numeric data should transfer between platforms.

***Is the TI-92 Plus allowed on College Board tests (SAT, ACT)?***
The TI-92 Plus is not allowed on standardized tests.

***Are there any support materials or workbooks for the TI-92 Plus?***
You can find a listing of support materials for the TI-92 Plus at the following URL.
http://education.ti.com/product/book/colbooks.html

***What are the appropriate courses for the TI-92 Plus?***
The Advanced Mathematics Software of the TI-92 Plus makes it a useful tool for Calculus I - Calculus III, Linear/Matrix Algebra, and Differential Equations.  Of course, the TI-92 Plus can also be used for Algebra, Geometry, and High School math courses, due to the existing TI-92 functionality.

***How much do the AMS upgrades for the TI-92 Plus cost?***
Advanced Mathematics Software upgrades for the TI-92 Plus are available from Texas Instruments at no charge.

***Is Texas Instruments still selling the TI-92 Plus Module?***
No. Texas Instruments is no longer selling the TI-92 Plus Module.

***Does the TI-92 Plus use the same ViewScreen as the TI-92?***
Yes. The TI-92 ViewScreen will work with the TI-92 Plus.

***What Data Collection tools is the TI-92 Plus compatible with?***
The TI-92 Plus is compatible with the CBL, CBL2 and the CBR.

***What are the main differences between the TI-92 and TI-92 Plus?***
The TI-92 Plus has Advanced Mathematics Software, additional memory, and upgradability.

***How much does the TI-92 Plus cost?***
Please contact your Instructional Products Dealer for pricing information:
http://education.ti.com/global/dealers.html

***What do Flash and electronically upgradable mean?***
Flash is a technology that allows the calculators software to be upgraded electronically. Using the TI-GRAPH LINK cable and the link I/O port, the software can be updated with the latest maintenance updates or new software versions.

***Does the TI-92 Plus work with the TI-Graph Link?***
Yes, the TI-92 Plus is compatible with TI-Graph Link cables.

### *Texas Instruments GraphLink FAQ*

http://education.ti.com/product/accessory/link/faqs/faqs.html

- Build my own link cable?
- Extra spaces in front of each program line - why?
- File size limitations - why can large files from my calculator not be opened?
- Graph Link (DOS) interferes with my mouse or other serial card.
- I cannot open a "protected" TI-82 program using my Mac Graph Link.
- Inserting lists into (exporting from) an Excel™ spreadsheet.
- Installing Graph Link fonts on my Mac
- Mac 5400 will not work with Graph Link - what to do.
- Macintosh PowerBooks and Graph Link - does it work?
- New or Updated Graph Link software - where can I get it?
- Power drained from the calculator by the cable?
- Can I use a Printer port instead of Modem port on Macintosh?
- Site license for the TI-GRAPH LINK?
- StudyWorks™ - how does it work with the Graph Link?
- System requirements for the Graph Link.
- Transmission errors with Graph Link (Mac version).
- Transmission errors with Graph Link (PC version).
- Why can't I open an edit-locked program using my Graph Link?
- Can I connect my Graph-Link to my iMac, or other Macintosh computers that have USB ports?
- Does the TI-Graph Link USB work with the TI-82 or TI-85?

***Build my own link cable?***
Questions have been asked frequently about the cost of the cable to link between personal computers and TI Graphics Calculators. A number of people have wanted details to build their own cable, perceiving it to be merely a matter of matching the RS-232 port TX and RX lines to the proper pins on the calculator plug.

Texas Instruments wants the interconnection of the Graphics Calculator to other devices to be as simple and cost effective as possible.  For this reason, the link port on the calculator is not designed like RS-232.  A RS-232 type port requires accurate timing and involves circuitry that consumes significant power. The calculator's link ports have two data lines and a ground, using simple CMOS digital levels in a self clocking protocol that does not require any precise timing, but can operate well above 9600 bits per second.  This allows the linking between two calculators to be accomplished at minimal cost and power drain.

In line with this strategy, our first link cable (the black one (*should be 'the gray one', Ed.*)), utilized relatively simple transistor circuitry and did not even use the RX and TX lines of the RS-232 port. Instead, it used the RS-232 "control" lines (2 for inputs and 2 for outputs) to operate the data lines. The reason we had to change away from this approach is that many Macintosh computers do not have all the necessary control lines available on their serial port connections. To make a MAC compatible connection we had to communicate through RX and TX and this required putting an eight-bit, crystal controlled oscillator, PIC microcontroller in the cable. This microcontroller communicates with the PC at a standard baud rate and converts each bit into the self clocking protocol to communicate to the calculator. The power for the cable is derived from the PC and doesn't impose significant drain from the calculator. Overall, this approach provides a reasonable cost link without any cost burden to the calculator user who does not use a link to personal computers.

Due to the specialized components and also because some parts of the design are necessary to ensure compliance with FCC radio frequency emission regulations, it is not practical for individual users to build their own cables on a general basis. We hope this explanation helps you understand why this is.

***Extra spaces in front of each program line - why?***
When you try to "Open" a group file on a Macintosh, you will receive an Information Window which shows all lines of each program inside the group file. If you copy this and paste it into a new program, you will see three extra spaces at the beginning of each line. To avoid these spaces, "Ungroup" the group file instead of Opening it.

***File size limitations - why can large files from my calculator not be opened?***
Part of the difficulty here is tokenized size vs text size. The maximum 'text' size for editing (in the TI-82 and TI-83 software) is about 32k bytes. Tokenization reduces the size to much less (probably ~22k). What this means is that keywords like 'Then', 'Disp', 'PlotsOn', etc. are 1 or 2 bytes tokenized vs 4 or more in 'text'.

If you have a 23k program on your calculator in tokenized form (i.e. you created it on your calculator), and then attempt to edit this using the Graph-Link, the editable text image probably exceeds the 32k text edit limit. Basically, this large of a program is un-editable with Graph Link.

The workaround is not an easy one. The program has to be split into smaller segments (subprograms) and each one edited separately.

***Graph Link (DOS) interferes with my mouse or other serial card.***
Graph Link works with the IBM standard communications protocols. Com1 and Com3 share an interrupt request (IRQ), and Com2 and Com4 share an IRQ. Com1 and Com3 use IRQ 4 while Com2 and Com4 use IRQ 3.

Example: If you have a mouse on Com1 and have the mouse driver software loaded, you will have trouble running Graph Link on Com3. Network cards can also cause such problems. You will need to eliminate the IRQ conflict to have both devices working at the same time.

***I cannot open a "protected" TI-82 program using my Mac Graph Link.***
The older version of the Graph Link software for Macintosh would not allow users to read or update a protected TI-82 program once it is saved. The new version of the Mac software has changed this feature. Now, users can protect a program to be sent to the TI-82, but the user can still read and update these programs from their computer.

The new software is now included in the Graph Link package.

***Inserting lists into (exporting from) an Excel™ spreadsheet.***
You can import data from an Excel spreadsheet by the following procedure. In Excel, open the file that contains the data you wish to export. Select File, Save As... and choose Text (Tab Delimited) from the "Save File as Type" box. You will select the directory and give the file a name. Then go to the Graph Link software.

In the TI-82 Graph Link software, go to File, Utilities, Import ASCII Data File. Select List and one of the list names. Now you need to find and select the file you saved earlier. After you select OK, you will be asked for a name for the new list file (you can just use the default name). Select OK again and the file will be saved as a TI-82 list. You can then go to File, Open and see the file or simply download it to your calculator. (The process is very similar in the other Graph Link software. In the TI-83, TI-86, and TI-92, you will find the Import option under Tools.)

You can export data from a TI-82 list to an Excel spreadsheet in a couple of ways. First you can open the list in the Graph Link software and select all the elements. (Edit, Select All) You can then Copy and Paste it into the Excel spreadsheet. You can also export the information using the Export Data File under File, Utilities. This procedure is very similar to the importing above.

***Installing Graph Link fonts on my Mac***
If you are running system 6.* then you will need a font mover (font D/A mover) which opens your SYSTEM FILE and places the fonts in it. This Font D/A Mover is a Mac utility usually found on the TIDBITS disk that came with the Mac.

If you are running system 7.* simply drag the fonts from the Graph Link disk to a closed SYSTEM FOLDER. The Mac should then automatically move the fonts to the users SYSTEM FILE or FONTS FOLDER (depending on the version of system 7 software).

***Mac 5400 will not work with Graph Link - what to do.***
The Graph Link draws power from the serial port, which is different from most serial devices. This appears to be causing a problem with the Mac 5400's. To correct the problem, you will need to unplug the Graph Link cable, reboot the computer and plug the cable in after the computer starts.

***Macintosh PowerBooks and Graph Link - does it work?***
Yes, the Graph Link will work with the Macintosh PowerBooks.

The difficulty is that PowerBooks have multiple functionality on a single comm port. The Graph Link will not re-configure system settings, controlled from Control Panels or Chooser.

To get Graph Link to work with a PowerBook (as noted in the Link85.user and Link82.user files in the FREEWARE folder) you need to:

- Disable AppleTalk/LocalTalk from the Chooser.
- Launch the 'Portable' control panel, disable the internal modem, and enable the external modem.

After making the above changes, launch the Graph Link application. Graph Link will now communicate with the link hardware/calculator, since the "printer port" is now a "modem port".

***New or Updated Graph Link software - where can I get it?***
Software for new models of calculators is made available through the WWW in beta test form during software development.

When development is complete, it is included in the TI-GRAPH LINK package. New software becomes available free through the WWW after release, and is also available on diskette in the U.S. and Canada from our TI at 800-TI-CARES. There is a charge for ordering the software from TI to cover handling costs.

***Power drained from the calculator by the cable?***

If the Graph Link cable is plugged into the calculator and not to the computer, it will drain power from the calculator. If the Graph Link cable is plugged into the calculator and computer but the Graph Link software is not running, the cable will also drain power from the calculator.

The Graph Link cable gets power through the DTR line of the serial port. However, it is normally off and the Graph Link software turns it on. Therefore, you should plug the cable into the calculator after the cable is attached to the computer and the software is running.

***Can I use a Printer port instead of Modem port on Macintosh?***

The TI-82 and TI-85 Macintosh Graph Link software has been updated to allow a user to download programs with the Printer port, as well as the Modem port (like the TI-92, TI-86, and TI-83 software). Users that need this option can download updated software.

***Site license for the TI-GRAPH LINK?***

A site license is not needed to put the software on several computers. The software that comes with the TI-GRAPH LINK may be installed on several computers as needed. Of course, a Graph Link cable is needed to communicate with a calculator. One cable may be moved from computer to computer as needed.

***StudyWorks™ - how does it work with the Graph Link?***

Steps to use the Graph Link with StudyWorks:

- Upload a list (or matrix) to computer via the Graph Link.
- Export list from Graph Link into an ASCII data file (*.PRN). (You will need to save it in the same directory as your StudyWorks file is located.)
- "READ" ASCII data file into StudyWorks worksheet ()

***System requirements for the Graph Link.***

The different system requirements for DOS, Windows and Mac can be found in the features zone.

***Transmission errors with Graph Link (Mac version).***

If you are receiving Tranmission Errors when using Graph Link, check the following:

- Ensure that the cable is pushed all the way into the calculator (This is probably the most common difficulty, and easily overlooked).
- Check that your computer meets Graph Link's system requirements.
- Ensure that you are plugged into the same communications port that you have set in the Graph Link Preferences (modem or printer)
- Try restarting your Mac with the Extensions turned OFF, as some extensions may cause conflicts with the port. You can do this by pressing and holding the SHIFT key while the Mac is restarting. If this solves the problem, then restart the extensions one by one until the problem one is found.
- If you are using the modem port, make sure the external modem is turned ON.
- Turn Appletalk OFF (Appletalk can be accessed by going to the Apple menu and selecting "Chooser".

***Transmission errors with Graph Link (PC version).***

If you are receiving Tranmission Errors when using Graph Link, check the following...

DOS:

- Ensure that the cable is pushed all the way into the calculator (This is probably the most common difficulty, and easily overlooked).
- Check that your computer meets Graph Link's system requirements.
- If you use a notebook computer, make sure the port you are using is turned ON. Secondly, try using Graph Link with the battery savings turned off. These battery conservation mechanisms often create difficulties with Graph Link.
- Ensure that the COM port that the cable is plugged into and the COM port selected in Setup match.
- If you have a mouse and mouse driver, they may be causing an IRQ (interrupt request) conflict with Graph Link. Try rebooting the computer with out the mouse and/or mouse driver loaded.
- Ensure that you do not have a device conflict, such as an internal modem using the same COM port that the Graph Link is using.
- Ensure that the IRQ for COM1 & COM3 is set to IRQ 4, and COM2 & COM4 are set to IRQ 3. (Graph Link requires these settings, while other serial devices might not.)
- With DOS 6.0 and higher, try rebooting the computer bypassing the autoexec.bat and config.sys by pressing F5 when computer says "starting MS-DOS".

Windows:

- Check the first 6 bullets above.
- Check that your computer meets Graph Link's system requirements.
- Make sure Windows has not changed the condition of the COM ports.
- For example, the DOS version may work while the Windows version does not. Windows port setting under the Control Panel could have changed the com port setups.
- Check the Startup group to see if any additional programs are being loaded along with Windows that might interfere with serial communication.

***Why can't I open an edit-locked program using my Graph Link?***
In order to open a write-protected program, you must have version 1.1 or later of the Macintosh software which allows you to edit-lock a program. To determine the version of the Macintosh software, you can click and hold the Apple symbol and then click on About TI-GRAPH LINK. If you have an early version of the software, you will need to get a newer version.

Note: The Windows software has always allowed users to update a protected 82 program.
The TI-83, TI-86, and TI-92 softwares have had this feature from the beginning as well.

***Can I connect my Graph-Link to my iMac, or other Macintosh computers that have USB ports?***
Yes. You can purchase a TI-GRAPH LINK USB cable to connect your TI graphing calculator to a Macintosh USB port. To place an order via phone, call 1-800-842-2737. You can also place an order through our online store.

You can connect the Gray TI-GRAPH LINK cable to a Macintosh USB port by using a USB to serial adapter.

USB-to-Serial adapters are available on the WWW. For more information on USB-to-Serial adapters you can visit the following web sites:

http://www.keyspan.com/
http://www.usbstuff.com/
http://www.entrega.com

http://www.belkin.com

NOTE

Texas Instruments has not tested the adapters supplied by these companies. Some TI-Graph Link users are reporting success in using some of these adapters. However, we do not make any claim that they will work with your calculator or that their use will not damage your calculator. USE AT YOUR OWN RISK.

***Does the TI-Graph Link USB work with the TI-82 or TI-85?***
No. The TI-Graph Link USB does not work with the TI-82 or TI-85. You can connect the Gray TI-GRAPH LINK cable to a Macintosh USB port by using a USB to serial adapter.

USB-to-Serial adapters are available on the WWW. For more information on USB-to-Serial adapters you can visit the following web sites:

http://www.keyspan.com/
http://www.usbstuff.com/
http://www.entrega.com
http://www.belkin.com

## [B.2] TI documentation

In a web page redesign in March 2001, TI has evidently removed the *EightySomething* newletters. I have a few of the old ones; email me if you would like a copy.

TI calculator manuals, called guidebooks, are available on-line. These are the links to download the manuals:

TI-92 guidebook:
http://education.ti.com/product/tech/92/guide/92guideus.html

TI-92 Plus module guidebook:
(no longer available; use the 89/92+ guidebook)

TI-89 guidebook:
(no longer available; use the 89/92+ guidebook)

Combined TI-89/TI-92 Plus, AMS 2.05
http://education.ti.com/product/tech/92p/guide/guides.html

TI GraphLink guidebook:
http://education.ti.com/product/accessory/link/guide/guides.html

## [B.3] Web sites

There are thousands of independent TI calculator web sites. These are the best. In my opinion, of course.

### Visual Mathematics Illustrated by the TI-92 and the TI-89

http://www.imaxx.net/~gdorner/visual/

http://206.67.72.201/catalog/np/feb00np/2-287-59685-2.html

These two sites describe an excellent mathematics book written by George Dorner, Jean Michel Ferrard, and Henri Lemberg, and published by Springer-Verlag. As far as I know, this is the only 'advanced' mathematics book for the 89/92+. From the description:

> *Here is a selection of basic and advanced mathematics, unified by common themes and supported by graphical and formal calculations provided by the TI-92, the TI-92 Plus, and the TI-89 graphing calculators.*
>
> *The topics of mathmatics covered are those of higher level university courses for students of mathematics, computer science, physics, engineering and other sciences...*
>
> *Topics covered are in the fields of classical analysis and linear algebra, and in the overlap of the two. Chapters are devoted to discrete dynamical systems, differential equations, Fourier series, and approximation and interpolation theory.*

This book is expensive, but worth it. If you don't care to buy it for yourself, you could ask your school or local library to purchase it. It is a thorough, involving and accessible introduction to many fascinating and useful areas of mathematics.

### Stephen Byrne's List of TI sites

http://www.rit.edu/~smb3297/ti/

A nice list of TI oriented sites categorized by math and engineering topics.

### Olivier Miclo's ti-cas site

http://www.ti-cas.org/

One of the premium math sites. Emphasis on calculus, trigonometry, matrices and polynomials. Too much good stuff for me to summarize!

### Roberto Perez-Franco's Symbulator site

http://sq.calc.org

Here you can get the most powerful programs available for the 89/92+. Symbulator; a symbolic circuit simulation program. Diffeq; Lars Frederiksen's differential equation solver, includes solutions to multiple differential equations. Advanced LaPlace: another solid piece of work by Lars which solves for LaPlace transforms and inverses. Also: Fourier transform programs, state space program, discrete Fourier transform and inverse. You can also get Lars' RPN program here, which implements an RPN interface for the HW1 89/92+.

**S.L. Hollis' TI-89/92 math site**

http://www.math.armstrong.edu/ti92/

Lots of very good math programs. Single and multi-variable calculus, linear algebra, differential equations, probability, Gaussian quadrature and special functions. Very, Very Good Site!

**Frank Westlake's site**

http://frank.westlake.org/ti/

Several calculus functions, including directional derivative, gradient, partial and total derivatives, multivariable limits, Taylor approximations for multi-variable functions. Notes on using Var-Link to provide fast program and function documentation. Internet sockets to send and receive email with your calculator. Number base conversions. Roman numeral conversions. Image editor. Remote control with scripts. Lots of utilities for converting text files, including BMP, RTF, PIC, TEXT and STR formats. You must disable Javascript in your web browser to use Frank's site.

**Jack Hanna's Iview site**

http://users.bergen.org/~tejohhan/iview.html

Get Jack Hanna's *Iview* program here. The program runs on a PC and converts graphics files to PIC variables which can be viewed and manipulated on the calculator.

**Rusty Wagner's site**

http://rusty.acz.org/

Go here for the Virtual TI emulator (VTI). This is PC software that emulates the calculator. You'll need to do a ROM dump from your calculator to run it. 'ROM dump' means uploading the calculator ROM image to the PC with a GraphLink cable. ROM images are not legally available by any other means.

The advantage to using the emulator is that you can test programs before downloading them to your calculator. This can save battery life.

**Techno-Plaza**

http://www.technoplaza.net/

A very complete comparision between the TI-89 and the HP49G, with little or no bias. Some math programs. Good assembly programming tutorials.

**SoftWarehouse advanced functions**

ftp://ftp.ti.com/pub/graph-ti/calc-apps/92/swh/

Advanced math functions for the 89/92+. Although this was written for the original TI92, and many of these functions are now built into the 89/92+, there is still lots of good stuff here. In particular, the code style is clever and efficient - you can learn a lot about programming by examining these programs.

TI redesigned the web site in March 2001. The little remaining description of this function packages is at *http://education.ti.com/product/tech/92/faqs/faq20595.html*

**TI calculator program archives**

http://education.ti.com/student/coll/down/archive.html

Programs written by users and submitted to TI. Includes the inferential statistics package for the 89/92+.

**TI 89 Users Group - TAMUK math club**

http://www.tamuk.edu/mathclub/

This is another top-tier site. Lots of math programs in all the usual categories, but also other programs in science, engineering and CBL/CBR. Does not appear to accept submissions right now. Some programs are original, some have been collected from other sources, and modified for various reasons.

**Bhuvanesh Bhatt's site**

http://triton.towson.edu/~bbhatt1/ti/

Several TI-92 PLUS advanced math programs, including: Cauchy principal value of an integral, a differential equation graphing utility, a multiple linear regression function, a special functions package, a tensor analysis package, a Christoffel symbol package, and a complex analysis graphing package. Bhuvanesh also has a special functions package that implements a great many special functions. You can also get E.W.'s 89/92+ EquationWriter (EQW) here. This site also hosts this tip list.

**Stuart Dawson's surverying software**

http://www.dawson-eng.demon.co.uk/nexus/

Surveying software for working surveyors. The more powerful versions are *not* free, but there is a free version with reduced functionality.

**Andrew Cacovean's tip list & examples site**

http://www.geocities.com/TI_TipList/

Andrew hosts the web version of this tiplist, and you can also get the current version of the 89/92+ wishlist there. Andrew also has some very good tutorials on using the 89/92+ to solve basic and more advanced problems.

**Kevin Kofler's web site**

http://ti89prog.kevinkofler.cjb.net/

This is where to go to get Kevin's extremely handy *autoaoff()* program, which disables the default alpha keyboard in dialog boxes on the TI-89. Kevin has also written programs to automatically close open parentheses on the entry line, map common functions to unused key combinations on the TI-89, balance chemistry equations, use log and semi-log axes on function plots, find exact solutions to cubic and quartic equations, make date calculations, manipulate grey-scale pictures and display calendars.

**Paul Pollack's TI-92 Number theory programs**

http://www.geocities.com/Hollywood/2979/ntheory.html

A small collection of number theory programs, including tests for primality, tests for probable primality, factoring by Pollards rho method and p-1 method, and fast square roots of large integers. Seems to be old but still useful.

**Michael Lloyds TI calculator programs**

http://www.hsu.edu/faculty/lloydm/ti/prgmtabl.html

This site includes many programs in algebra, trigonometry, statistics and calculus. Some specific programs include Molleweide's equation for checking triangles, conic equation graphing, probability distributions (binomial, F, Pearson-Moment correlation, Student's T, chi-squared), ANOVA and number base conversions.

**Bubu's TI-92 programs**

http://www.multimania.com/bubuw/index_e.html

Only one program on this site, an implementation of Conway's 'game of life' cellular automata simulator. What makes this notable is a large collection of 'starting patterns' for the simulation.

## [B.4] Bibliography of TI-89 and TI-92 Plus books

This bibliography lists books related to the TI-89, TI-92 or TI-92 Plus calculators. Some are out of print, but you may find them at used book stores or web sites, or on eBay. Many of these books are little more than (very) expensive pamphlets: make sure you know what you're getting before you buy them.

For some of the books I give contents and excerpts from prefaces or introductions, to give you a better idea of what the book is about. If given, prices are in US dollars as of May, 2002.

The last section of the bibliography lists general calculator books not specifically related to TI calculators, but they may be interesting or useful.

The bibliography is organized in these sections:

Introductory and programming
General Mathematics
Algebra
Calculus and differential equations
Statistics
Geometry
Engineering
CBL, CBR
Other calculator books

### Introductory and programming

*Getting Started with the TI-92/TI-92 Plus*
Carl Swenson, Brian Hopkins, John Wiley and Sons, 1998

*Graphing Calculators: Quick and Easy Using the TI-92 Plus*
David P. Lawrence, Pencil Point Press, Inc., ISBN 1-881641-46-5

*Great TI-92 Programs - Vol 1.*
Bernhard Kutzler, David Stoutemeyer, Teachware Series, 1997, ISBN 3-901769-00-5

*Introduction to the TI-89 (Part 1)*
Bernhard Kutzler, Eric Armin Inc., ISBN 3-901769-14-5

*Introduction to the TI-89 (Part 2)*
Bernhard Kutzler, Eric Armin Inc., ISBN 3-901769-15-3

*Introduction to the TI-92*

Bernhard Kutzler, Eric Armin Inc., ISBN 3-901769-02-1

*Learning Programming with the TI-92: Structures and Techniques*
Steve Blasberg et al, Texas Instruments, 1997, ISBN 1-886309-08-6
(TI Explorations Series, High school and university, $22.50)

"In this book, we introduce many of the programming commands and functions of the TI-92. As you use this book to learn programming, you also will learn structured programming techniques that help you program efficiently in the TI-92 programming language and in other programming languages as well. These techniques will help you write well-documented programs that are easy to follow, modify, and maintain."
Contents: Introduction to the TI-92. The Action Block. Top-Down Programming. The Decision Block. The Repetition Block. Working with Data. Working with Parameters and Functions. Working with Menus and Dialog Boxes. Working with Graphics. Geometry on the TI-92. Solutions to Practice Problems. Index.

*Quick and Easy Reference Guide*
David P. Lawrence, Pencil Point Press Inc., ISBN 1-58108-033-6

## General Mathematics

*Discovering Math on the TI-92*
Chris Brueningsen et al, Texas Instruments, 1996, ISBN 1-886309-05-1
(TI Explorations series, High school algebra through calculus, $16.00)

"The purpose of this workbook is to provide a tool for teachers as they first use the TI-92 and consider ways to incorporate this technology into their classrooms. Our goal is to provide innovative and practical examples of how to begin using the new features of the TI-92 in high school math classes. The workbook is designed to meet the needs of a wide variety of students. Accordingly, many different areas of mathematical content are covered, ranging from algebra to advanced calculus.

"The activities are written assuming the reader has no TI-92 knowledge. Easy-to-follow instructions with step-by-step keystrokes are included for every activity. The book is presented in worksheet format so students can record their answers as they work. Extensions and extra practices are included at the end of many exercises, allowing students to investigate advanced topics related to concepts presented in the activity."

Contents: Exploring the Unit Circle. Getting Down to Basics. Solving Systems of Equations. Using Linear Programming in Real Life. What Is a Linear Regression?. Warming Up to Heating Curves. Evaluating Rational Function. Figuring Areas. Graphing An Extra Dimension. Functioning on Your Own. Modeling Damped Motion. What is the Number "e"?

*Exploring Precalculus Mathematics with the TI-89/TI-92/TI-92 Plus*
Michael Schneider, Gilmar Publishing, ISBN 0-888808-04-7

*Learning Mathematics Through the TI-92*
William Lawrence, Philip Yorke, ISBN 0-86238-489-3

*Mastering the TI-92: Explorations from Algebra through Calculus*
Nelson Rich et al, Gilmar Press, ISBN 0-9626661-9-X

*Visual Mathematics Using the TI-92 and TI-89*
G.C. Dorner, J.M. Ferrard, H. Lemberg, 2000, Springer France Editions, 440pp, ISBN 2-287-59685-2

Contents: Discrete dynamical systems. Differential equations. Fourier analysis. Interpolation and approximation. Orthogonality. Eigenvalues and eigenvectors. Calculator guide. Bibliography. List of the programs. Symbols used in the book. Index.
(Highly recommended by DAB.)

## Algebra

*Advanced Algebra with the TI-89*
Brendan Kelly, Brendan Kelly Publishing, ISBN 1-895997-12-7

*Algebra 1 Explorations and Applications; Activities for TI-81, TI-82 and TI-92 Calculators*
McDougal Little Publishing

*Algebra 2 Explorations and Applications; Activities for TI-81, TI-82 and TI-92 Calculators*
McDougal Little Publishing

*Investigating Advanced Algebra with the TI-92*
Brendan Kelly, Brendan Kelly Publishing, ISBN 1-895997-05-4

*Skill and Practice Masters in Algebra Using the TI-89*
David P. Lawrence, Pencil Point Press, Inc., ISBN 1-58108-041-7

*Skill and Practice Masters in Algebra Using the TI-92*
David P. Lawrence, Pencil Point Press, Inc., ISBN 1-881641-56-2

*Solving Linear Equations with the TI-92*
Bernard Kutzler, Eric Armin Inc., ISBN 3-901769-03-X

*Solving Systems of Linear Equations with the TI-92*
Bernard Kutzler, Eric Armin Inc., ISBN 3-901769-05-6

## Calculus and Differential Equations

*Advanced Placement Calculus with the TI-89*
Ray Barton, John Diehl, Texas Instruments, 1999, ISBN 1-886309-27-2
(TI Explorations series, high school calculus, $16.00)

"This book presents an introduction to the numeric, graphic and analytic features of the TI-89. It is our hope that as teachers and students become familiar with these features, they will experience the same excitement all mathematicians feel when a new idea is discovered."

Contents: Functions, Graphs, and Limits. Differentiation. Applications of the Derivative. Integration. Riemann Sums and the Fundamental Theorem of Calculus. Applications of Integrals. Differential Equations and Slope Fields. Parametric, Vector, Polar, and 3D Functions. Infinite Sequences and Series. Appendices: TI-89 Keystrokes and Menus. Common Calculus Operations. Creating Scripts. Solutions to the Exercises. Index.

*AP Calculus with the TI-89 Graphing Calculator*
George Best, Richard Lux, Venture Publishing, 1999, ISBN 1-886018-25-1

*Applied Calculus With Technology*
Richard C. Weimer, Brooks/Cole Pub Co., 1998, 784pp, ISBN 0534354491, $86.36

From the publisher:
"Understanding that technology can be both a valuable tool and as an active companion in the learning of calculus, Weimer has produced a textbook that students-those majoring in business, management, economics, and the social, life and physical sciences-will appreciate for the way it helps guide them into the 21st century. Students are introduced to functions and associated preliminary algebraic material, and then are presented with basic concepts of differential calculus. The organization and careful introduction of material is designed to help even poorly prepared students succeed. This text is ideal for professors who wish to integrate DERIVE® or the TI-92 graphing calculator into the applied calculus course."

Contents:
1. Functions. 2. The Derivative. 3. Graphs and Applications of the Derivative. 4. The Derivative: More Properties. 5. Derivatives of Logarithmic and Exponential Functions. 6. The Integral. 7. Additional Integration Topics. 8. Calculus of Several Variables. Appendix A: Important Derive? Commands for IBM-PC's or PC Compatibles. Appendix B: Derive? for Windows, Version 4.02. Appendix C: TI-92. Answers to Odd-Numbered Exercises.

*CalcLabs for the TI-92*
Selwyn Hollis, Brooks/Cole Publishing, 1997/98, ISBN 0-534-34970-6
(Not found on Brooks/Cole web site)

*CalcLabs With the TI-92 for Stewart's Calculus: Concepts and Contexts, Single Variable*
Selwyn Hollis, Brooks/Cole Pub Co., 1997
(Not found on Brooks/Cole web site)

*CalcLabs With the TI-92 for Stewart's Mulitvariable Calculus: Concepts and Contexts*
Selwyn Hollis, Jeff Morgan, Brooks/Cole Pub Co., 1998
(Not found on Brooks/Cole web site)

*Calculus TI 92 Lab Manual*
Ron Larson, Houghton Mifflin College, ISBN 0-395-90062-X
(According the Houghton Mifflin web site, this product is available 'with' various calculus texts. Prices are given for Windows and Mac versions, so perhaps it is actually software or a PDF file - impossible to tell from the description)

*Calculus with the TI-89*
Brendan Kelly, Brendan Kelly Publishing, 1999, 96pp, ISBN 1-895997-13-5, $16.95

*Exploring Calculus and Differential Equations with the TI-89/TI-92/TI-92 Plus*
Michael Schneider, Gilmar Publishing, ISBN 1-888808-06-05

*Insights into Calculus Using TI Calculators: 83 Plus, 86, 89, 92, and 92 Plus*
Robert Smith, Roland Minton, McGraw-Hill Higher Education

*Insights into Calculus Using TI Calculators :*
*83 Plus, 86, 89, 92, 92 Plus for Use With Calculus : Premiere*
Robert T. Smith et al, McGraw Hill College Division

*Introduction to the TI-92: 37 Experiments in Precalculus and Calculus*
Charles Lund, Edwin Andersen, MathWare, ISBN 0-96223629-7-2

*Investigating Calculus with the TI-92*
Brendan Kelly, Brendan Kelly Publishing, ISBN 1-895997-07-0

*Scripting Guide for the TI-92 and TI-92 Plus: PreCalculus and Calculus Applications*
J. Douglas Childs, Texas Instruments, 1998, ISBN 1-886309-20-5
(TI Explorations series, High School and University Pre-Calculus and Calculus, $16.00)
"The TI-92 provides instructors with a broad array of new teaching options. You can de-emphasize algebraic manipulation and, at the same time, focus on more powerful methods of solving problems using mathematics and the TI-92. This guide is designed for instructors, but is also useful for students, since it contains scripts and activities for students. By reading sections of this guide, students can gain insights about learning mathematics that they would not obtain from textbooks. This guide can be a valuable resource for teaching and learning calculus and precalculus with the TI-92 Plus.

"This guide has several purposes:

- To present some ideas about how to use the TI-92 to do calculus and precalculus calculations and processes.
- To make instructors aware of this new scripting feature in computer algebra graphing calculators.
- To describe a large number of learning activities for students. I recommend that you modify the scripts and activities to suit the needs of your students and your particular situation.
- To help mathematics instructors develop a pattern for using this somewhat complicated machine in a minimally obtrusive way.
- To help instructors think about what their students are actually learning and to provide an environment that may suggest some alternative instructional goals.
- To help students effectively use a TI-92 to learn and to do calculus and precalculus.

"Remember that this guide is designed to provide ideas and an opportunity to make improvements in your calculus course. Tools like the TI-92 will eventually change the way students learn mathematics - how and when this actually occurs is up to you. This guide is more of a beginning than an end. Most scripts and activities need to be modified by individual instructors to achieve their particular goals."

Contents: Getting Started with Scripts. Foundations. Functions. Applications. Limits. Derivatives. Integrals. Sequences and Series. Differential Equations. MultiVariable Calculus. Appendix.

*Skill and Practice Masters in Calculus Using the TI-89*
David P. Lawrence, Pencil Point Press, Inc., ISBN 1-58108-040-9

*Skill and Practice Masters in Calculus Using the TI-92*
David P. Lawrence, Pencil Point Press, Inc., ISBN 1-881641-91-2

*Teachware Series, Support in Learning:*
*Optimization - Graphically, Numerically and Symbolically with the TI-92*
ISBN 3-901769-20-X

*Ti-92 Lab Manual for Calculus*
Ron Larson, John Musser, Houghton Mifflin College

**Statistics**

*Advanced Placement Statistics with the TI-89*
*Extensions for Advanced Placement Statistics with the TI-89*
Larry Morgan, Roseanne Hofmann, Charles Hofmann, Texas Instruments, 2001

PDF format. This books uses the *Statistics with List Editor* Flash Application
(TI Explorations series, $22.50, *Extensions*: $9.00)

"With the help of the Statistics with List Editor Application, the TI-89 brings to life topics in statistics such as: histograms, categorical data, probability and simulation, binomial distributions, confidence intervals, and more."

Contents: Part 1, Exploring Data: Graphical Displays of Univariate Data. Summarizing Distributions of Univariate Data. Comparing Distributions of Univariate Data. Exploring Bivariate Data. Describing Categorical Data. Part 2, Planning a Study: Randomization in Sampling and Experiments. Part 3, Probability with Simulations: Probability as Relative Frequency, The Normal Distribution, Sampling Distributions,  Sampling Distribution of Differences. Part 4, Statistical Interface: Confidence Intervals. Tests of Significance. Special Cases of Normally Distributed Data. Appendices: Installing the Statistics with List Editor Application. Quick Reference for the Statistics with List Editor Application. Index.

(*Extensions* is available in electric form only at http://epsstore.ti.com.)

"This book gives additional topics that could be used in a first or second course in statistics, including Analysis of Variance, Multiple Regression, Forecasting, and Nonparametric methods. "

Part 5: Other Probability Distributions. Inference for the Standard Deviation and Variance of Normal Populations for One and Two Samples (The F Distribution). Analysis of Variance (ANOVA), One- and Two-Way. Multiple Linear Regression. Forecasting. Some Non-parametric Procedures. More Fits to Bivariate Data. Appendices: Installing the Statistics with List Editor Application. Quick Reference for the Statistics with List Editor Application. Program Listings for exsmooth and forecast. Index.

*Investigating Statistics with the TI-92*
Brendan Kelly, Brendan Kelly Publishing, ISBN 1-895997-06-2

*RS-92 Resampling Stats for the TI-92*
Bill Quails, Resampling Stats Inc., 1997.

*Statistics and Probability with the TI-89*
Brendan Kelly, Brendan Kelly Publishing, ISBN 1-895997-14-3

*The Practice of Statistics: Ti-83/89 Graphing Calculator Enhanced*
Daniel S. Yates et al, Publishes June 2002

**Geometry**

*92 Geometric Explorations on the TI-92*
Michael Keyton, Texas Instruments, 1996, ISBN 1-886309-06-X
(TI Explorations series, high school geometry, $22.50)

"This book provides a variety of student activities using Geometry on the TI-92. Activities ranging from preliminary discovery of basic theorems typically seen in a high school course to advanced explorations show students exciting geometric results.

"The activities are arranged in a logical sequence for a high school syllabus, but are not in the order of any specific textbook. They have been designed as independent of each other as possible. Thus, you can reorder them to fit any sequence."

Contents: Segment Partition (Between) and Midpoint. Distance from a Point to a Line (Circle). Angle Partition (Addition) and Angle Bisector. Vertical Angles. Perpendicular Bisector. Perpendiculars and Parallels.   Linear Pair and Perpendicular Pair. Bisectors of Two Angles That Form a Linear Pair. Angle Bisector (Characterization). Circle (Radius, Diameter, Chord). Triangle: Medians and the Centroid. Isosceles Triangle. More About the Isosceles Triangle. Angle Inscribed in a Semicircle. Altitude, Median, and Angle Bisector of a Triangle. Parallel Lines and Angles. Triangle Sum. Equilateral Triangle and a Curiosity. Perpendicular Bisectors: Circumcenter and Circumcircle. Three Points Determine a Circle (sometimes). Triangle: Altitudes and the Orthocenter. Pedal Triangle of a Point and a Triangle. Angle Bisectors: Incenter and Incircle. Midsegments and the Medial Triangle. Tangents to a Circle. Trisecting a Segment. Hinges. Geometric Mean. Animation and Locus. Right Triangle; Making a Table. Using a Table to Make Conjectures. Arc Measure Macro. Inscribed Angle and Angle Formed by Two Chords. Angle Formed by Two Secants or a Secant and a Tangent. Angle Formed by a Tangent and a Chord or by Two Tangents. Steiner's Power of a Point / Chord-Chord. Steiner's Power of a Point / Secant-Secant / Secant-Tangent. Equilateral Triangle and Square. Parallelogram. Isosceles Trapezoid. Rectangle. Rhombus. Kite. Cyclic Quadrilateral. Varignon Quadrilateral. Curiosity of an Isosceles Triangle, Rectangle, Isosceles Trapezoid. Pythagorean Theorem. Midvexes of a Parallelogram. Perpendicular and Parallel Angles. Intersecting Circles. Parallel Lines and Proportions. Parallel Through a Side of a Triangle or Through a Circle. Angle Bisector of a Triangle. The Nine-Point Circle. Trisection Polygons. A Maximum and a Minimum in One Problem. Midvex of a Parallelogram and Isosceles Triangle. Tangents to Two Circles. Angle Bisector Revisited. Triangles with Parallel Sides (Homothecy). Distance From a Point to a Line (Circle Triangle) to a Point. Dilations (Homothecy Revisited). Similar Triangles Are Everywhere. Trigonometric Tables. The Medial Triangle Revisited (Homothecy). The Euler Line. Circumcircles, Reflections, and (...). Antipedal Point and Triangle. Harmonic Conjugates. Menelaus. Ceva. Menelaus, Ceva, and Harmonic Conjugates. Quadrilateral and Harmonic Conjugates. Pappus. Desargues. Pascal's Mystic Hexagram. Miquel. Carnot. Centroids Galore and What Point Is It Anyway? Wallace Line. Tritangent Circles. Gergonne. Nagel. Feuerbach: The Tritangent Circles and the Nine-Point Circle. Isotomic and Isogonal Conjugates. Bisectors of Interior and Exterior Angles of a Triangle. Maltitudes, Diacenter, and Medcenter. Cyclic Quadrilateral Curiosities. Butterfly. Brocard Points. Orthocenter and Circumcenter. Orthocenter Revisited. Teacher Information. References.

*Geometrical Investigations for the Classroom on the TI-92*
Charles Vonder Embse, Arne Engebretsen, Texas Instruments, 1996, ISBN 1-886309-04-3
(TI Explorations series, high school geometry, $16.00)

"The explorations in this book are intended to promote an environment of inquiry through thought provoking, open-ended problems. There is no intended order of topics. Any exploration can be used where it fits into your program. There is no suggested time for an exploration. Some students may think about a given problem for weeks or months, coming up with new insights and interconnections as they consider other problems. It may be appropriate to present one exploration, or part of one, each grading period as a group or individual project.

"No matter how you use these activities, the emphasis should be on exploration - trying anything and everything that comes to mind. Once the discoveries are made, ask students to explain why they are true. Depending on the level of the students, these explanations could be anything from statements of the findings to detailed proofs. But remember, like real mathematics in the making, proof should be the final step in the discovery process, not the first."

Contents: Investigating Properties of Kites. Investigating Properties of Trapezoids. Investigating Properties of the Diagonals of Quadrilaterals. The Orthocenter of a Triangle. Cyclic Quadrilaterals. Tessellations and Tile Patterns. Polygons and Vectors. The Simson Line. Investigating Properties of Lines in the Plane. Construction Tools: Interactive Function Graphs.

*Geometry Explorations and Applications, Activities for the TI-81, TI-82 and TI-92 Calculators*
McDougal Littel Publishing

*Skill and Practice Masters in Geometry Using the TI-92*
David P. Lawrence, Pencil Point Press, Inc., ISBN 1-58108-020-4

## Engineering

*Electrical Engineering Applications with the TI-89*
David R. Voltmer, Mark. A. Yoder, Texas Instruments, 1999, ISBN 1-886309-25-6
(TI Explorations series, university electrical engineering, $22.50)

"To Students:

"This book is written for electrical engineering students. It is a collection of examples that show how to solve many common electrical engineering problems using the TI-89. It is not a textbook; if you do not know how to solve the problem, look it up in your textbook first. If you do know how to solve the problem, this book will show you how to use the TI-89 to get the answer with more insight and less tedium. We show you how to use the TI-89 in class, in lab, on homework, and so forth.

"To Instructors:

"When writing this book, we resisted the temptation to show how the TI-89 can be used to solve problems in ways that differ from standard electrical engineering texts. Although it has the power and ability to approach many problems in new ways, that was not our focus. Our focus is to help students learn the basic material better by showing them how to use the TI-89 to do the tedious things so they don't get lost in the details. Our approach was best summed up by Gottfried Wilhelm Leibniz when, in the 17th century, he said, "It is unworthy of excellent men to lose hours like slaves in the labor of calculation." "

Contents: DC Circuit Analysis. Transient Circuit Analysis: Symbolic. Transient Circuit Analysis: Numeric. Steady-State Circuit Analysis and Filter Design. Power Engineering. Laplace Analysis: The s-domain. The Convolution. Fourier Series. Vectors. Vector Calculus. Electromagnetics. Transmission Lines. Antennas. Manipulating Lab Data: The Diode. Financial Calculations. Index.

*Engineering and Technical Math on the TI-92: Electronic and Electrical Applications*
Charles R. Adams, David Hergert, Texas Instruments, 1997, ISBN 1-886309-10-8
(TI Explorations series, High school and university electrical engineering, $22.50)

"This book was written to show the power of the TI-92 for electrical and mechanical students at the vocational and two year technical level. Also included in the text are highlighted sections appropriate for engineering students.

"The goal of the authors is to help students customize the TI-92 for electrical applications. The TI-92's rich set of mathematical and engineering features make it far superior to any other calculator of its kind. The instructions in this book on formula entry, graphing, data collection and display, the CBL utility and report writing will help students and technicians fully utilize the TI-92's capabilities.

"This electrical volume shows the most commonly used formulas for electrical technicians and engineers. It works as a great supplement to textbooks on circuit analysis and electronics.
Some of the features of the TI-92 are highlighted through examples, including:

- Introduction of vectors and phasors using-the powerful Vector tool from the Geometry menu.
- Use of PopUp list boxes which allow students to select colors for the resistor color code, and material selection for resistivity.
- Use of the powerful solve utility to rearrange Ohms Law and the Power formula. This should be of great use to high school students not yet adept at algebra.

- Use of a function that parallels resistors. This function allows students to enter complex series-parallel circuits in a simple format.
- A complete set of base conversions (including fractional) for digital electronics.
- Graphical display of a charging capacitor.
- Display and analysis of two and three phase waveforms, similar to the method used to calculate phase shift on an oscilloscope.
- Use of functions to calculate the impedance of capacitors and inductors. These functions allow students to enter complex AC impedance in a simple one-line format.
- A description of how readings can be directly entered into the calculator while an experiment is being performed. A DC motor is used as an example.

"All of the examples are geared toward helping students understand the power and convenience of the Tl-92. Many of the examples would be more difficult or impossible if attempted on another calculator.

Contents: Chapter 1, Basic Use of the TI-92: Solving Equations. Entering Numbers. Display Digits Mode. Inserting and Over-Typing. Storing and Recalling Variable Values. Keyboard Shortcuts. Pretty Print Mode. Displaying the Home Screen. Graphing a Function. Exponents and Powers of Ten. Coordinate Conversions. Chapter 2, DC Circuit Analysis: Ohm's Law. The Power Formula. Current and Charge. The Resistivity Equation. The Resistor Color Code. Capacitor Charging. Parallel Resistance. Collecting and Displaying Motor Data. Chapter 3, AC Circuit Analysis: The Sine Wave. Capacitive and Inductive Reactance. Solving Series AC Circuits. Solving Parallel AC Circuits. Average Value. Average Value (Sine Wave). RMS Calculations. Chapter 4, Digital Electronics: Base Conversions. Converting from Binary to Decimal. Converting From Decimal to Binary. Converting Binary to Hex. Converting Hex to Binary. Converting from Hex to Decimal. Converting Decimal to Hex. Binary Addition. Binary Subtraction. Boolean Operations. Chapter 5, Analog Electronics: Diode Circuit Operating Point. Common Emitter Amplifier. Chapter 6, Power Distribution: Y to Delta and Delta to Y Resistance Conversions. Three-Phase Circuits. Three-Phase Power. Chapter 7, Magnetism and Electromagnetic Fields: Magnetic Flux. Induced Voltage in a Magnetic Field. Force Generated in a Magnetic Field.

**CBL, CBR**

*TI-92 and CBL Lab Manual: Precalculus Applications*
Glencoe Publishing, 1996, ISBN 0-02-824316-1

*Microchemistry for the TI-92/CBL*
Tom Russo, Theta Technologies, ISBN 1-888167-05-X

**Other calculator books**

*Handbook of Electronic Design and Analysis Procedures using Programmable Calculators.*
Bruce K. Murdock, Van Nostrand, 1979, 525pp.

*TI-59 and HP-41CV Instrument Engineering Programs*
Stanley W. Thrift, Gulf Publishing Co., 1983, 366pp.

*Scientific Analysis on the Pocket Calculator*
Jon M. Smith, John Wiley & Sons, 1977 (2nd ed), 446pp, ISBN 0-471-03071-6
Contents: Introduction to pocket calculator analysis. Numerical evaluations of functions on the pocket calculator. Advanced analysis on the pocket calculator. The programmable pocket calculator. Financial analysis for engineers and scientists. Appendices: Some tricks of the pocket calculator trade, Matrix analysis on the pocket calculator, Complex numbers and functions, Formulas for commonly encountered calculations.
(Highly recommended by DAB)

*Advanced Applications for Pocket Calculators*
Jack Gilbert, Tab Books, 1976, 304 pp. ISBN 0830658246

*Calculator Programs for Chemical Engineers*
Vincent Cavaseno, ed, 1982.

*Microwave Circuit Design Using Programmable Calculators*
J.W. Allen, M.W. Medley, Artech House, 1980, ISBN 0-89006-089-4
Procedures for performing S-Parameter calculations with HP and TI programmable calculators.

*Statistics By Calculator: Solving Statistics Problems With The Programmable Calculator*
Peter W. Zehna, Prentice-Hall, 1982, 308 pp, ISBN 0138448116

*Engineering Statistics with a Programmable Calculator*
William Volk, McGraw-Hill, 1982, 362pp, ISBN 0-07-067552-X
Contents: Introduction. Statistical parameters. Probability distributions. The t test. Chi square test. Variance and the analysis of variance. Regression. Appendices: Hewlett Packard calculator programs, Texas Instruments Calculator programs. Programs for HP-97 and TI-59.

*Mathematical Astronomy With A Pocket Calculator*
Aubrey Jones, Wiley, 1978, 254pp, ISBN 0-470-26552-3
Contents: Time. Precessional constants for selected epochs. Reduction for precession. Reduction from mean to apparent place. Proper motion. Sun, moon and planets. Visual binary star orbits. Ephemerides of comets. Approximations. Appendices: Visual binary star orbits. Programs for calculators using RPN logic. Index.

*Practical Astronomy with your Calculator*
Peter J. Duffet-Smith, Cambridge University Press, 1989, 200pp.

*Illustrated Pocket Programmable Calculators In Biochemistry*
John E. Barnes, Alan J. Wring, John Wiley & Sons, Inc., 1980, 363 pp.
For use with Hewlett-Packard HP-67/97 and Texas Instruments TI-58/59.
Contents: Aqueous solutions of Small Molecules. Macromolecules In Solution. Sedimentation. Ligand Binding and Kinetics. Thermodynamics In biochemistry. Spectroscopy. Isotopes In Biochemistry. Appendices. Index.

*Phaselock Loops for DC Motor Speed Control*
Dana Geiger, Wiley, 1981, 206pp, ISBN 0471085480
Uses TI-59 programs

*Reservoir Engineering Manual.*
Reuven Holo, Haresh Fifadaro, Penn Well Books, 1980.
27 reservoir engineering programs. Uses TI-59 programs.

*Drilling Engineering Manual*
Martin E. Chenevert, Reuven Hollo, Penn Well Books.
A manual, based on use of the TI-59 programmable calculator or similar ones for petroleum engineers covering nearly all phases of drilling.

*Synthetic-Hydrograp Computations on Small Programmable Calculators*
Thomas Croley, Iowa Institute of Hydrologic Research, 1980, ISBN 0874140153.

*Astro-Navigation Calculator: A Handbook for Yachtsmen*
Henry Lesion, David & Charles, 1984, 112pp.

"Written especially for yachtsmen, this book shows how the traditional methods of obtaining an astro-navigational fix can be superseded by using a scientific pocket calculator and a new book of calculator tables."

*Calculator Navigation*
Mortimer Rogoff, Norton, 1979, 418pp, ISBN 0-393-03192-6
Contents: Calculators and navigation. Coastwise navigation. Sailing. Celestial navigation. Loran. Appendices: Recording procedures, Customized programs, Setting decimal and trigonometric mode on the HP-67 / HP-97, Non-print operation of the HP-97, Interrupting the display interval on the HP-67, Using the HP-41C, Program listings. Index.

*Radar Calculations Using the TI-59 Programmable Calculator*
William A. Skillman, Artech House, Inc. 1983, 405pp, ISBN 0-89006-112-2.
Contents: Preface, Antennas, Propagation, Detection Probability, Signal-To-Noise Calculations, Filters and Filtering, Receivers and Processing, Appendices:. Basic Operations, Extended Operations, Calculator Clubs and Books, Master Library Subroutines, Translation from TI-59 to Basic, List of Acronyms and Abbreviations, List of Symbols, Index.

## Appendix C: Glossary

This section defines some terms I use in the tips. Some definitions may seem obvious to English-speaking users, but the calculator community is thoroughly international. The glossary incorporates (with permission) many entries from Ray Kremer's Graphing Calculator glossary.

**@1...@255**
Represents an arbitrary constant, where distinct constants are distinguished by the numbers 1, 2 and so on, appended to the @ symbol. The constant may not be an integer; see @n1 below for comparison. Arbitrary constants may occur in solutions returned by *zeros().* The first arbitrary constant created is called @1. Subsequent constants are numbered from 2 to 255. The suffix resets to 1 when you use *ClrHome*.

**@n1...@n255**
Represents an arbitrary integer, where distinct integers are distinguished by the numbers 1, 2 and so on, appended to the @n. Arbitrary integers may occur in the solutions to some equations, for example, in the expression @n1 + 7, @n1 may be replaced by any integer. The first arbitrary integer which is created is called @n1. Subsequent integers are numbered from 2 to 255. After the 255th integer is created, numbering continues with @n0. You may use [F6] Clean Up 2:NewProb to reset arbitrary integers to @n1.

**2D**
Acronym for 'two dimensional'. Refers to the calculator graphing mode in which functions one independent variable are plotted, for example, y=f(x)

**3D**
Acronym for 'three dimensional'. Refers to the calculator graphing mode in which functions of two independent variables can be plotted, for example, z = f(x,y).

**68K, 68000**
Refers to the microprocessor used in the TI-89 / TI-92 Plus, which is a Motorola MC68000.

**89**
Abbreviation for TI-89

**92, 92+, 92p**
Abbreviations for TI-92 Plus, although *92* strictly refers to the original TI-92, not the TI-92 Plus.

**algebraic entry**
Method of typing an expression on the calculator in much the same way that expressions are shown in text books. Compare to *RPN*.

**AMS**
TI's acronym for *advanced mathematics software*. This refers to the basic operating software (operating system) for the TI-89 / TI-92 Plus calculators. Various revisions are identified by the version number, for example, AMS 2.05. Some bugs, fixes and tips only apply to certain AMS versions. TI seems to be abandoning the term AMS in favor of *base code*.

Only the most recent version of the AMS is available for download on the TI web site, but TI will e-mail older versions on request. Your e-mail service must be able to handle the AMS as an attachment, which is over one megabyte. Assembly programs are often incompatible with newer AMS versions.

These are known AMS versions:

| | |
|---|---|
| 1.00 | Original version for TI-92 with Plus module. Also HW1 TI-89 |
| 1.01 | TI-92 Plus, HW1 TI-89 |
| 1.05 | TI-92 Plus, HW2 TI-89 |
| 2.01 | Not an official release; leaked out on a few TI-89s in Europe |
| 2.03 | Added flash application support |
| 2.04 | Changed assembly program size limit from 8K to 24K |
| 2.05 | Fixed a bug in 2.04. The most current version. |
| 2.06 | Rumored version number for upcoming Voyage 200 Personal Learning Tool. Adds icon flash application menu and real-time clock support |

**annunciator**
A legend in the display status line, which shows particular states of the calculator. For example, some annunciators are RAD, DEG, BUSY and BATT.

**ans**
Refers in general to answers (results) to previous entries. The *ans* items are shown on the right-hand side of the history display.

**"anything but math"**
A TI contest in early 2002, the purpose of which was to contribute classroom activities designed to use TI products outside of math classes. Roughly coincided with the announcement of the Voyage 200 Personal Learning Tool. Since the contest required submission of classroom plans and other teacher-specific activities, it was not really intended for student participation.

**APD™**
Automatic Power Down™. A calculator feature: the calculator automatically turns itself off (powers down) after some time interval. The APD delay can be adjusted with ASM programs; see the archives at ticalc.org. The APD will be defeated on some IBM-compatible PCs if you leave the black GraphLink cable connected, but close the GraphLink software.

**Approx mode** See *Auto, Exact, Approx modes*

**archive, unarchive**
*archive* means to move a variable from RAM to flash memory. *unarchive* means to move a variable from flash memory to RAM. Archive may also refer to the flash memory itself.

**argument**
An argument is a number or expression which is supplied as input to a program, function or command. For example, in the expression sin(x), *x* is the argument.

**asm, ASM**
An abbreviation for 'assembler' or 'assembly language'. 'Assembler' may also refer to the program which converts assembly program source code to object code. Assembly is the lowest level language in which the calculator can practically be programmed. Assembler allows the fastest possible programs, as well as direct access to the calculator hardware. However, carelessly written assembly programs can cause crashes which disable the calculator, or worse, cause incorrect operation. C programs are also considered assembly programs, since only the assembled object code is on the calculator. See also *machine code*, *shell* and *no-stub*.

**ASM limit**
This is an artificial limit to the size of compiled programs that can be run on the TI-89 / TI-92 Plus. TI enforces this limit to prevent software piracy (theft) of a few large, for-pay applications. The ASM limit unfortunately also severely limits the program functionality that can be implemented by independent developers. The ASM limit was originally 8K bytes, but was later increased to 24K bytes.

**assembler, assembly** See *asm*.

**Auto, Exact, Approx modes**
The modes in which the TI-89 / TI-92 Plus can be set, to evaluate expressions. In Exact mode, all calculations are performed with exact values for constants. In Approx mode, all calculations are performed with floating-point approximations of numeric constants. In Auto mode, the calculator attempts to use the 'best' mode (either Exact or Approx) to evaluate the expression. If an expression includes constants with decimal points, Approx mode will be used. The mode is set in the Mode screen, which is displayed when the [MODE] key is pressed.

**Automatic Power Down™.** See APD.

**Auto-paste**
A calculator home screen feature. Use [UP] and [DOWN] to highlight an entry or answer in the home screen, then press [ENTER] to paste the expression to the current cursor position in the entry line.

**auto-repeat**
The cursor movement feature by which the cursor will automatically move if the cursor keys are pressed and not released.

**auto-simplify**
The process by which the calculator automatically simplifies symbolic expressions. Expressions are simplified to put them into a standard form. Auto-simplification may sometimes be desired, but often it makes some algebraic operations difficult or impossible. Auto-simplification cannot be disabled on the TI-89 / TI-92 Plus.

**backup, back-up**
A complete copy of the memory of the calculator that is stored on a PC. May also refer to the act of creating a backup. GraphLink software is used to create a backup. *Restoring a backup* means to copy the backup from the PC to the calculator.

**backup battery**
The lithium coin cell in the TI-89 and TI-92 Plus which maintains RAM contents if the main batteries discharge or are replaced.

**base code**
An alternative TI term for AMS; see *AMS*.

**benchmark**
A program or test case with the sole purpose of evaluating some aspect of the performance of the calculator or computer. Accuracy, correctness, speed or code size may be tested.

**binary**
Refers to numbers expressed in the base-2 number system. The TI-89 / TI-92 Plus can represent integers in base-10 (decimal), binary and hexadecimal (base-16).

**black-bar** See *crash*.

**boolean**
Applies to variables or conditional (logical) expressions which evaluate to values of *true* or *false*. *True* and *false* are boolean constants. A boolean expression combines boolean constants and variables with boolean operators and evaluates to a boolean constant. Boolean logic is named in honor of the 19th century English mathematician George Boole, who first formalized this logical system.

**brick**
Rarely-used slang which refers to the TI92 or TI92+ calculator, because of its large size and weight, compared to the TI-89.

**bug**
A flaw or fault in software which causes incorrect results. More specifically, a program has a bug, by definition, if it returns results which do not meet the specification for the software.

**built-in**
Describes functions or features which are available in the calculator AMS, without installing other programs or applications.

**C**
A high-level language used to write calculator programs. C programs can be faster than TI Basic programs, and can directly access the calculator hardware and internal operating system functions. C programs must be written and compiled on a PC, then the object code is downloaded to the calculator for execution. Two C compilers are available: TIGCC is a third-party free-ware compiler, and TI also supplies a C compiler with the SDK.

**Cabri Geometry**
A flash application for the TI-89 and TI-92 Plus which allows editing and analyzing Euclidean geometrical constructions. Also available as PC-based software as Cabri Geometry II. Cabri Geometry is free for the TI-92 Plus but not for the TI-89. Cabri Geometry was developed by TI and supports language localization. See also *Geometer's Sketchpad*.

**CAS**
An acronym for *computer algebra system*. This term refers to the calculator software specifically concerned with performing symbolic calculations, as opposed to purely numeric calculations.

**CBL™**
Acronym for 'calculator-based laboratory'. The CBL (a TI product) is a hardware device which connects to the TI-89/TI-92 Plus (and other calculators) through the GraphLink port. The CBL provides data acquisition functions and can measure voltage, temperature, frequency and other physical parameters.

**CBR™**
Acronym for 'calculator-based ranger'. The CBR (a TI product) is a hardware device which connects to the TI-89 / TI-92 Plus (and other calculators) through the GraphLink port. The CBR uses ultrasonic ranging to measure the distance between itself an the target.

**certificate**
A file which is generated by TI. Signed applications (see *signing*) require a certificate so that they can run on the calculator. See also *SDK*.

**clipboard**
Calculator memory in which cut or copied expressions are temporarily stored. The clipboard contents can be pasted.

**command**
A routine which is neither a function nor a program. All commands are built-in. The arguments for a command are not passed in parentheses, unlike the arguments for programs and functions. See also *program* and *function*.
**command line** See *entry line*

**complex**
Applies to an expression or number which has an imaginary component. For example, 3 + 2i is a complex number, where $i = \sqrt{-1}$ . *Complex* does not mean 'complicated'.

**conditional operator**
An operator such as >, < or =, which is applied to two expressions. A conditional expression evaluates to *true* or *false*.

**constraint**
An expression that limits the range of operation of expression evaluation. Constraints are applied with the 'with' operator |. For example, in the expression *a+b*|*a=2*, the constraint is *a=2*. Constraints may be composed with various combinations of Boolean operators.

**contrast**
Refers to the LCD screen contrast, that is, the level of darkness. The contrast changes with battery voltage. New batteries result in a dark screen, but the contrast decreases as the batteries age. The contrast is adjusted by pressing the [DIAMOND] key simultaneously with the [+] and [-] keys, as indicated by the green legend on the keyboard. If the contrast is set too light, it may appear that the calculator is off when it is actually on. Oddly, the contrast can be set on the VTI emulator, and must be changed when the emulator is reset.

**copy** See *cut, copy, paste*

**crash**
An error condition in which the calculator will not respond to keystrokes, perform calculations, or turn on or off. Also called freeze-up, lockup or black-bar. The 'black-bar' term comes from the symptom of a thick black bar appearing at the top of the LCD display when a crash occurs. Crashes are usually caused by ASM games and shells. Crashes from normal calculator operation or TI Basic programs are extremely rare. The display often shows the error message *Address Error* after a crash. Instructions for recovering from a crash are given in the User's Guides.

**current**
Adjective which describes a default calculator object. For example, the 'current folder' is the folder in which variables are stored, if a different folder is not specified. The 'current matrix' is the matrix that is currently in the Data/Matrix editor.

**cursor**
The display symbol which indicates the position at which text appears when keys are pressed. The cursor symbol is shown as a flashing vertical bar, unless *insert* mode is active, in which case the cursor is shown as a flashing solid rectangle.

**cursor keys**
The keys which move the cursor in the display. Also called the 'arrow keys'. In this document I refer to these keys as [UP], [DOWN], [LEFT] and [RIGHT]. The TI-89 has dedicated cursor keys, the TI-92 Plus has a blue cursor pad.

**cursor pad** See *cursor keys*

**cut, copy, paste**
Calculator text editing features. *Cut* removes the marked expression and places a copy in the clipboard. *Copy* places a copy of the marked expression in the clipboard. *Paste* inserts the clipboard contents at the cursor location.

**Data/Matrix Editor**
The application built into the calculator which is used to create or edit data variables, matrices and lists. It is accessed with the [APPS] key.

**DBus**
The name of the GraphLink serial link as described in the SDK manual. So named because the data lines are labeled D0 and D1?

**Derive**
A trade name for a PC-based computer algebra system, from which the TI-89 / TI-92 Plus CAS was derived. TI now owns Derive and continues to sell it.

**DG**
Acronym for discussion group.

**discussion group**
An electronic messaging system provided for users by TI on their web site. Discussion groups are similar to Usenet news groups, but the TI discussion groups are moderated by TI to remove trolls, vulgarity and profanity, and offers to buy or sell. There are dozens of TI discussion groups, focusing on different calculators and fields of study.

**ebay**
Abbreviation for the web auction site www.ebay.com. An inconvenient way to buy used calculators at prices near those of brand new units.

**emulator**
An emulator is software that runs on a PC which simulates the operation of the calculator, including running programs. Only one third-party emulator, called VTI, is available. To use the emulator, you must have a calculator ROM image on the PC. You violate TI copyrights if you use a ROM image with the emulator and you do not actually own the calculator.

**entry line**
The line at the bottom of the calculator display where commands and instructions are entered.

**EOS; EOS hierarchy**
A TI acronym for Equation Operating System. The EOS specifies the order in which operators are applied. The TI-89 / TI-92 Plus Guidebook specifies the EOS hierarchy in Appendix B. Note that exponentiation is applied before negation, and that logical operators (and, or, etc.) are also prioritized.

**Exact mode** See *auto, exact, approx modes*

**Equation Writer** See *EQW*.

**EQW**
Acronym for equation writer. An equation writer is a program which enables you to enter expressions in Pretty Print format, in contrast to the usual command line format. This is similar to the equation editors in MS Word, or WordPerfect, but calculator equation writers actually perform math. A programmer who

goes by the pseudonym E.W. has written an equation writer for the TI-89 / TI-92 Plus. Two versions are available: a free version, and a for-pay flash application. The flash application includes additional features and functions.

**errornum**
A system variable which contains the number of the most recent error. *errornum* is zero if no errors have occurred, or if ClrErr has been executed.

**expression**
A combination of variables and, optionally, operators and functions. For example, *a+b*, *sin(x)* and *y* are all expressions.

**false**
A built-in system variable which indicates the Boolean 'false' value.

**file extension**
This is a three character suffix on the names of files stored on a personal computer. The following table shows the file extensions for the TI-89 and TI-92 Plus.

| TI-89 | TI-92 Plus | Description |
|---|---|---|
| n/a | .9xa | Geometry figure (constructions) |
| .89c | .9xc | Data variable |
| .89d | .9xd | GDB (Graph database) |
| .89e | .9xe | Expression (numbers, equations, etc) |
| .89f | .9xf | Function (Y-vars, user-defined functions) |
| .89g | .9xg | Group |
| .89i | .9xi | Picture |
| .89k | .9xk | Flash Application Software |
| .89l | .9xl | List |
| .89m | .9xm | Matrix |
| .89p | .9xp | Program |
| .89q | .9xq | Certificate |
| .89r | .9xr | Lab report |
| .89s | .9xs | String variable |
| .89t | .9xt | Text variable |
| .89u | .9xu | Operating system |
| n/a | .9xx | Geometry macro |

**flash application or flash app**
An application that can run directly from flash memory, without first being copied into RAM. Flash applications can only be created with the TI SDK and by obtaining a certificate from TI. Certificates are not free; TI charges for them. Flash applications are started from the [DIAMOND] [APPS] key.

**flash; flash ROM**
*flash* is a short way to say flash EEPROM, which is one of the two types of memory on the TI-89/TI-92 Plus: the other type of memory is RAM. 'EEPROM' is an abbreviation for electrically erasable programmable read-only memory. You can store programs and variables from RAM to flash with the archive operation, which frees up RAM for calculations. The contents of FLASH can be quickly read, but it takes more time to write to the flash memory. The contents of flash memory are retained if the batteries are discharged or removed. See also *RAM*.

Flash memory has a limited (but large) number of erase/write cycles. The rating for the TI flash memory is a conservative 100,000 cycles. An erase cycle occurs only when a *garbage collection* is done, or when you upgrade the AMS. Under normal use, the flash will last for many years.

'Flash' can also be used as a verb: to *flash the calculator* or *flash the ROM* usually means to upgrade to a new operating system.

**floating-point**
A method used by calculators (and computers) to represent numbers with non-zero fractional components. A variety of methods and number bases can be used, but, in general, a number is represented as a signed mantissa implicitly multiplied by a signed exponent. Floating-point calculations are inherently imprecise, since some numbers cannot be exactly represented with a finite number of bits.

**flood**
A large number of messages posted to a discussion group with malicious intent to displace other valid messages.

**folder**
A named area of memory in which variables are saved. The calculator always has at least one folder called *main*. You may create your own folders. It is not possible to create a folder within a folder. One folder is always the *current* folder. The name of the current folder is shown in the display status line.

**friendly window**
A graph window with settings that result in pixel coordinates with 'round' values. This means that the pixel coordinates of the graph cursor are values such as 1.5, 1.51, 1.52, instead of 1.499, 1.54999, 1.51999, etc.

**freeze-up** See *crash*

**function**
A routine with zero, one or more arguments which may return a result to the home screen or to a calling routine. See also *program* and *command*.

**function key**
The blue keys labeled [F1] to [F8] on the TI-92 Plus, and [F1] to [F5] on the TI-89. On the TI-89, [F6], [F7] and [F8] are accessed with the yellow [2nd] key. The operations performed by the function keys are determined by the current operating mode of the calculator, and usually select functions in the toolbar.

**garbage collection**
A process that the calculator automatically performs to reorganize the contents of flash memory, which packs the variables into memory more efficiently. When you delete a variable from flash memory, it is not deleted at that time, it is just marked as 'not used'. The variable still occupies flash memory space. When the flash memory is full, the calculator operating system performs a garbage collection. Variables marked as deleted are actually erased at this time. This process is similar to defragmenting a PC hard drive. Refer to the *TI-89 / TI-92 Plus Guidebook* for more details.

**Geometer's Sketchpad**
A flash application for the TI-89 and TI-92 Plus which allows construction and analysis of geometrical models. Geometer's Sketchpad was developed by KCP Technologies Inc, and does not support language localization. See also *Cabri Geometry*.

**global variable**
A variable which is stored in a folder. Global variables can be recalled to the history display, and

programs and functions can also read the contents of global variables. See also *local variable*.

**GraphLink; GraphLink cable**
'GraphLink' can refer either to the physical cable that connects the calculator and the PC (or two calculators), or the software that runs on the PC. The GraphLink cables are not free; the GraphLink software is free and can be obtained from the TI web site. The GraphLink cable is not the same as the unit-to-unit cable included when the calculator is purchased.

There are four different versions of the GraphLink cable: the gray GraphLink, the black GraphLink, the USB GraphLink, and third-party cables. The gray GraphLink was the original cable. The black GraphLink is the new cable, and it can transfer data much faster than the gray cable. The USB cable is only available for Apple computers.

The GraphLink software is available for IBM-compatible PCs and Apple computers. The purpose of the software is to transfer variables and programs between the calculator and the PC. GraphLink software can also be used to back-up and restore the calculator's memory, and to edit TI Basic source code.

The GraphLink connector is the small round connector on the calculator into which the GraphLink cable is plugged.

**graph screen**
The screen on which function graphs and data plots are shown.

**group file**
Calculator files which have been combined into one file with the GraphLink software.

**hand key**
The TI-92 Plus has the hand key; the TI-89 does not. It is located above function key [F5]. It is used with [UP] and [DOWN] to scroll large expressions.

**hardware version** See *HW1*, *HW2*.

**hexadecimal, hex**
Refers to numbers expressed in the base-16 number system. The TI-89 / TI-92 Plus can represent integers in base-10 (decimal), base-2 (binary) and hexadecimal.

**history, history display, history area**
The area of the home screen in which previous calculations are results are shown. The history display is directly above the entry line.

**home screen**
The default display screen on which calculations and commands are entered, and calculated results are shown. The home screen is one of several display screens used by the calculator. Other screens are the graph screen and the program I/O screen. The home screen is displayed by pressing [HOME] if some other screen is currently shown.

**HW1, HW2**
Acronyms for *hardware version 1* and *hardware version 2*. The original TI-89 and TI-92 were built with HW1. More recent TI89s and the TI92+ are built with HW2. HW1 TI-89s are hard to find now. HW2 calculators run about 20% faster than HW1 calculators. HW2 calculators have different display driver hardware that also improves performance. HW2 calculators also have built-in hardware which TI uses to enforce the ASM limit.

**HP**
An acronym for Hewlett-Packard, another respected vendor of fine graphing calculators.

**implied multiplication**
A convention used by the TI-89 / TI-92 Plus CAS which allows omission of the multiplication operator in some expressions. For example, *2x* is equivalent to *2*x*. However *a(x)* is not interpreted as *a*x*, but instead as calling function *a()* with an argument of *x*. Also, the expression *2ab* is not interpreted as *2*a*b*, but instead as *2*ab*, where *ab* is a single variable.

**integer**
A number with no fractional component. -1, 0 and 42 are integers; -1.1 and 2/3 are not integers. See also *floating-point*.

**interpolation**
Estimating the value of a function between two known values.

**k or K**
Abbreviation for 1024. 1K bytes is 1024 bytes.

**kernel**
Software which gives assembly programs access to calculator machine resources. Assembly programs which are not no-stub require a kernel. See also *shell*.

**keyboard shortcut**
A key press combination to perform a calculator function. The keyboard shortcuts are listed in the *TI-89/TI-92 Plus Guidebook*.

**LCD**
An acronym for liquid-crystal display. The displays used on the TI-89 / TI-92 Plus are LCDs.

**LCD connector; LCD port**
The unlabeled rectangular connector on the back of the TI-92 Plus which is used to connect an LCD display screen for overhead projection. Actually called the ViewScreen port. The TI-89 does not have this connector. See also *ViewScreen*.

**LSD; LSB** See *most significant digit*

**least significant digit; least significant bit** See *most significant digit*

**library**
A calculator file containing ASM utility programs which are used by other programs. Libraries save memory when several calculator programs need the same utilities, since the library eliminates duplicating the utilities in each program. Libraries can cause problems if programmers do not ensure that the library version is compatible with the program version, or the library is not located in the correct folder.

**linear format**
The opposite of 'Pretty print': a representation of an expression such that all the symbols and operators are on one line.  Expressions are shown in linear format in the entry line. The expression (a^b+b^)^(1/3) is in linear format.

**Linux**
An open-source operating system alternative to Windows. There is some Linux support for TI-89/TI-92 Plus calculators, including a TIGCC C compiler and file transfer software which performs GraphLink functions.

**local variable**
A local variable is created by a function or program, and only exists while the program is running. Local variables are used for temporary storage during program execution. See also *global variable*.

**lockup** See *crash*

**machine code**
Strictly, program code for the calculator expressed as binary integers. Machine code is the lowest level of programming, assembly language (with mnemonics) is the next level up, and high-level languages such at TI Basic or C are the highest programming level. Assemblers and compilers convert language instructions to machine code. Sometimes *machine code* is used when *assembly* is meant.

**manual**
Instructions for the calculator. TI no longer includes complete manuals with the TI-89 and TI-92 Plus, instead, an abbreviated edition is included. Complete manuals, which are necessary for most calculator operation, are available for download at www.education.ti.com. Bound, printed editions can also be purchased from TI, in a variety of languages, but these are usually not as up to date as the downloaded versions.

**math class**
From Ray Kremer's glossary: "The source of many concepts used by the calculators. Useful for determining what entry will achieve the desired result".

**mode**
A means to control different aspects of calculator operation. The mode settings are shown with the [MODE] key.

**model family**
A group of calculator models with similar hardware, software or compatibility. The TI-89, TI-92, TI-92 Plus and the (unreleased) Voyage 200 form a model family.

**modifier key**
A key which modifies the function of other keys. The modifier keys are [2nd], [DIAMOND], [SHIFT], [alpha] (TI-89 only) and [HAND] (TI-92 Plus only).

**most significant digit**
Refers to the digit in a number with the greatest positional magnitude. For example, in the number 12345, '1' is the most significant digit. May be abbreviated MSD. The *least significant digit* is the digit in the number with the least positional magnitude; in the number 12345, '5' is the least significant digit, or LSD. For binary (base-2) numbers, the most significant digit is called the most significant bit (MSB), and the least significant digit is called the least significant bit (LSB).

**MSD; MSB** See *most significant digit*

**multiplication, implied** See *implied multiplication*

**native assembly**
Describes assembly programs which need no shell. See also *ASM*.

**no-stub**
Describes an assembly or C program which can (or must) be run without a shell. I have suggested that programs which are *not* no-stub be called 'stubby' programs, but this hasn't caught on. The term *no-stub* comes from the C source code directive which results in object code not requiring shell support.

**numeric**
Used to describe operations on number, as opposed to operations on symbols. See also *symbolic*.

**on-line store**
The TI web page (http://epsstore.ti.com/) where you can download software including AMS versions, the GraphLink software, and free and for-pay applications. You can also buy some calculators and accessories.

**order of operations**
Operation priority is specified by the operating system. The priorities are

1. Parentheses ( ), brackets [ ], braces { }
2. Indirection (#)
3. Function calls
4. Post operators: degrees-minutes-seconds (°,','"), factorial (!), percentage (%), radian ($^r$), subscript ([ ]), transpose ( $^T$ )
5. Exponentiation, power operator (^)
6. Negation (-)
7. String concatenation (&)
8. Multiplication (*), division (/)
9. Addition (+), subtraction (-)
10. Equality relations: equal (=), not equal (≠ or /=), less than (<), less than or equal (≤ or <=), greater than (>), greater than or equal (≥ or >=)
11. Logical not
12. Logical and
13. Logical or, exclusive logical xor
14. Constraint “with” operator (|)
15. Store (→)

**over-clock**
A modification to the calculator processor clock circuit which increases the operating frequency and therefore increases the calculator operating speed. Since the TI-89 / TI-92 Plus clock speed is set with a capacitor, over-clocking involves replacing the capacitor with one of smaller value. Over-clocking voids the warrantee and reduces battery life. If taken to extremes, over-clocking can also cause faulty operation, crashes and GraphLink transmission problems.

**parameter**
A constant which controls the operation of a function or feature. Most often refers to the settings in the *Window* screen, which controls the appearance of graphs. Sometimes also used to mean *argument*.

**paste** See *cut, copy, paste*

**PC**
Acronym for personal computer. Usually refers to an IBM-compatible computer, unless Apple computers are specifically mentioned.

**pixel**
A single point or 'dot' on the calculator display.

**PLT**
TI's acronym for Personal Learning Tool, which they use to describe the Voyage 200. Although the Voyage 200 is an updated TI-92 Plus, TI does not specifically call it a graphing calculator. See also *Voyage 200* and *v200*.

**port, porting (software)**
*Porting* is the process of rewriting a program to run on a calculator other than that which it was originally intended. For example, TI-83 programs cannot run on the TI-89. Ports of popular programs are common.

**post**
A discussion group message.

**Pretty print**
The graphical method of displaying expressions in which the expression elements are shown as they are usually written by hand or set in type. For example, fractions are shown with the numerator above the denominator, and exponents are shown slightly above the other elements.

**program**
A routine which has zero, one or more arguments. Programs cannot return results to calling routines or to the home screen. See also *command* and *function*.

**program archive**
A large collection of calculator programs on the internet, either on a web site or FTP site. Used to download other people's programs, and to distribute your own.

**program I/O screen**
The screen on which programs display text and expressions. One of several calculator display screens.

**QWERTY**
Alpha keyboard as used on the TI-92 Plus and the Voyage 200. So named because of the arrangement of the letters QWERTY on the top row.

**RAM**
One of two types of memory in the calculator; also see *flash*. RAM is an acronym for random access memory. Variables and programs are stored in RAM, unless you store them into flash memory by archiving them. Variables can be quickly stored into RAM and read from RAM. The contents of RAM can be lost if the main batteries are removed *and* the RAM lithium back-up battery is discharged.

Usually, programs can only run in RAM. An archived program (in the flash memory) is first copied to RAM before execution. However, *flash applications* are executed directly from flash memory.

**reserved function name**
The TI-89 / TI-92 Plus reserve some names for system variables and reserved function names. These names are specified in the *TI-89 / TI-92 Plus Guidebook*, in Appendix B. You cannot use these names for your own variables.

**ROM**
1) The integrated circuit (microchip) on which the calculator operating system (AMS) is stored. ROM is an acronym for Read-Only Memory. The TI-89 and TI-92 Plus use flash ROM. 2) The calculator operating system itself, as in the ROM version.

**ROM version**
A number which specifies the version of the calculator operating system software. Changes are made to the OS software over the product life of the calculator, and the ROM version number identifies the particular version on a given calculator. The OS changes may include added features or functionality, bug fixes, or software changes to comply with hardware changes. Assembly programs must often be rewritten to accommodate a new ROM version. The ROM version can be displayed by pressing [F1] at the home screen, then [A] About. The ROM version is specified as "Advanced Mathematics Software Version xxx", where "xxx" is the ROM version number. Also see *AMS*.

**routine**
A general term which refers to programs, functions and commands. Routines may be built-in to the calculator, or may also be written in assembly, TI Basic or C.

**RPN**
An acronym for Reverse Polish Notation. This is a calculator operating method in which the expression operands are entered on a stack, then operations are performed on the stack elements. Hewlett-Packard calculators use RPN; TI calculators do not. Independent programmers have written RPN interfaces for the TI-89 / TI-92 Plus. RPN is more keystroke-efficient than other interface methods; complicated expressions can be entered in fewer keystrokes without parentheses. More description of RPN can be found at the HP Museum calculator site (http://www.hpmuseum.org/).

**scroll**
To move the display contents so that hidden parts of an expression are shown. Long or large expressions will not fit in the small calculator display, and you need to scroll the expression to see all of it.

**SDK**
Acronym for Software Development Kit. The SDK is a combination of software and documentation with which you can write applications for the TI-89 / TI-92 Plus. The software consists of a C compiler and assembler. Flash applications can only be developed with the SDK. The SDK is available from TI in two versions; one version is free. With the free version, you cannot create flash applications.

**self-test**
Software built into the calculator to test operation during manufacturing or repair. While the self-test software is not documented by TI, some user have discovered certain parts of it. The self-test usually results in reset memory.

**serial port**
The RS232 serial port on a computer. This is not a USB port. One version of the GraphLink cable is compatible with serial ports.

**shell**
An alternative operating environment which supports assembly programs. Shells are not developed or supported by TI. Shells were originally needed to run assembly programs on the TI-89 and TI-92 Plus. Assembly programs which are properly coded do not need shells; see *no-stub*. A shell is essentially an assembly program used to run other assembly programs.

**signing**
The process by which TI turns a program into an application which runs on the calculator.

**silent linking**
TI's term for the capability of GraphLink communications without using the Link menu. Older TI calculators do not have silent linking.

**SMAP II**
The name of the BCD floating point number format used on the TI-89 and TI-92 Plus. Refer to the *TI-89 / TI-92 Plus Developer Guide* (the SDK manual) for more details.

**source code**
The original, human-readable form of calculator programs. Can refer to programs in assembly, C or TI Basic. The source code is converted to assembly code with an assembler or compiler. In the case of TI Basic, the source code is converted to tokens by an on-calculator interpreter. These tokens in turn invoke built-in assembly language segments on execution.

**status line**
The line at the very bottom of the calculator display, which shows the current folder and various annunciators.

**symbolic**
Used to describe operations on variables as opposed to numeric calculations. For example, *a+b* is a symbolic calculation, but *1.2 + 3* is a numeric calculation.

**symbolic constant**
A constant which can be manipulated by the CAS as a symbolic value, instead of simply a numeric value. Symbolic constants include π, *i,* *e* and ∞.

**syntax error**
Message displayed by the calculator when a program does not have the proper syntax, that is, the program does not conform to the rules of the programming language. The syntax error dialog box usually includes an option to open the program editor at the offending line of code. To fix a syntax error, you must understand both the intent of the program and the programming language.

**system variable** See *reserved function name*

**text editor**
The built-in TI-89 / TI-92 Plus application in which you create and edit text variables.

**test operator** See *conditional operator*

**TI**
Texas Instruments (what else?)

**TI92+**
Acronym for TI-92 Plus.

**TI Basic**
The BASIC programming language interpreter which is built into the TI-89 / TI-92 Plus. Often just called BASIC.

**ticalc, ti-calc**
Refers to the web site www.ticalc.org, a popular third-party site for TI calculator information and programs. Heavily oriented towards games, and accepts all program submissions, regardless of

repetition with existing programs. Not affiliated with TI, but probably the most popular, well-known TI enthusiast site.

**TI-cares**
A TI expression for calculator customer support. You may telephone, mail or email TI-cares.

**ticas, ti-cas**
Refers to the web site http://www.ti-cas.org/, a popular third-party site for TI calculator information and programs. Has both French and English pages. Oriented towards education, mathematics and practical applications, not games. Unfortunately, ti-cas is not available at the time of this writing because of a disagreement between TI France and the web site author.

**TI Connect**
TI software designed to replace the GraphLink software. Provides file transfer and supports the USB GraphLink cable, but does not yet include a TI Basic program editor. TI Connect is only compatible with flash-based calculators, including the TI-89 and TI-92 Plus.

**TIGCC**
Acronym for TI Gnu C compiler, which is a C cross-compiler for the TI-89 / TI-92 Plus based on the Gnu compiler collection. TIGCC is a third-party compiler, not a TI product. TIGCC is a popular alternative to TI's SDK. For more information, see http://tigcc.ticalc.org/.

**tios; TIOS; TI-OS**
Acronym for 'TI operating system'. Describes the built-in operating system which controls the operation of the calculator. See also *AMS*.

**tokenize**
The process by which TI Basic programs are converted to tokens. Tokens are short constants which represent TI Basic commands and functions. Programs are tokenized the first time the program runs.

**toolbar**
The toolbar is the set of menu tabs that are displayed at the top of the screen, and are accessed with the function keys [F1] -[F8]. The toolbars change depending on which screen is shown, and you can create your own custom toolbars.

**troll**
A person who posts to a discussion group for the primary purpose of annoying or inciting other members of the group. May also refer to the post itself.

**true**
A built-in system variable which indicates the Boolean 'true' value.

**UI**
Acronym for user interface, which is the visible system through which you specify input to the calculator, and the calculator provides the results.

**unarchive** See *archive, unarchive*

**underscore character**
The character "_".  Type it with [DIAMOND] [MODE] on the TI-89, and [2nd] [P] on the TI-92 Plus. Used to specify a measurement unit (_kg) or complex symbolic variables (var_).

**unit-to-unit cable**
Cable used to connect two calculators of the same model family for file transfers. Included in the package with the calculator. It is not a GraphLink cable and cannot be used for file transfer with a personal computer.

**USB**
Universal Serial Bus. A relatively new computer port which is replacing serial and parallel ports on PCs. New Apple Macintosh PCs have only USB ports. A USB GraphLink cable is available.

**v200**
Voyage 200

**ViewScreen, ViewScreen port**
LCD display panel for overhead projectors which displays the calculator screen. TI-92, TI-92 Plus and Voyage 200 calculators all have a built-in ViewScreen port; a special version of the TI-89 is available with a ViewScreen port. Each ViewScreen port support all the calculators in a model family.

**Voyage 200**
An updated TI-92 PLUS with more flash memory, an optional icon-based user interface for flash applications, and a real time clock. TI calls this a 'personal learning tool' (PLT) instead of 'graphing calculator', with the intent that it be used in classes other than math. The Voyage 200 is physically smaller than the TI-92 Plus and comes with a USB GraphLink cable. The Voyage 200 will be available in the fall of 2002. See also *PLT*, *v200*.

**VTI**
Acronym for Virtual TI; a calculator emulator. See *emulator*.

**window screen**
The screen which is shown when the [WINDOW] key is pressed. This screen shows the parameters that control the appearance of graphs and data plots.

**'with' operator**
The binary operator indicated by the vertical bar character '|'. The 'with' operator is entered with [2ND] [K] on the TI92+, and with the dedicated [|] key on the TI-89.

**Y= Editor**
The built-in editor which is used to enter and edit the y-functions. The Y= Editor is displayed by pressing [Y=].

**Zip file**
A PC file which contains one or more compressed PC files. Most calculator programs are distributed as zip files, and you must use an unzip utility (such as WinZip or PKzip) to unzip the files before sending them to the calculator.

# Appendix D: Command Quick Reference Guide

The following quick reference guide is a condensation of the command reference in the *TI-89/TI-92 Plus Guidebook*. It consists of the following sections:

- Summary of commands
- Reserved system variable names
- EOS (Equation Operating System) Hierarchy

I have included this information in the tip list to save you the trouble of referring to the guidebook while using the tip list. It may also be useful in its own right, if printed out at reduced size to be carried with your calculator.

Note that the codes for *setMode()*, *getMode()*, *setGraph()* and *setTable()* are shown in tables in the respective function definitions. These tables include the more recent numeric string codes.

## Quick reference guide to functions and commands

The function or command name is shown in **bold** text as the first line of the description. The lines immediately following show the arguments, if any. Arguments are shown in *italic* text. Optional arguments are shown in square brackets []. *expr* is an expression, *var* is a variable. If a function can accept more than one type of argument, they may be shown on one line separated by commas, for example,

(*expr*), (*list*), (*matrix*)

If a function may optionally take no arguments, that is indicated as (). Program and function structures may be shown with statements separated with the colon :, which is the symbol actually used for that purpose in programs. Functions and commands indicated with symbols are shown at the end of the reference guide.

**abs()**
(*expr*), (*list*), (*matrix*)
Return absolute value of real argument or modulus of complex argument.

**and**
*expr1* and *expr2*
*list1* and *list2*
*matrix 1* and *matrix 2*
Return *true* or *false* or a simplifed form.

*integer1* and *integer2*
Bit-by-bit 32-bit integer compare. Arguments larger than 32-bit signed values are reduced with symmetric modulo operation. Mode must be Auto or Exact.

**AndPic** *picVar [, row, column]*
Logical-AND of graph screen and *picvar* at (*row, column*); default is (0,0)

**angle()**
*(expr), (list), (matrix)*
Return angle of arguments interpreted as complex numbers. Undefined variables are treated as real variables.

**ans()**
*(), (integer)*
Return answer from home screen. *integer* can be 1 to 99; not an expression; default is zero.

**approx()**
*(expr), (list), (matrix)*
Evaluate argument as decimal value if possible.

**Archive** *var1[,var2] [,var3] ...*
Move variables to flash ROM.

**arcLen()**
*(expr,var,start,end)*, *(list,var,start,end)*
Return arc length of *expression* (or each element of *list*) with respect to *var* from *start* to *end.*

**augment()**
*(list1,list2)*
Return list which is *list2* appended to *list1*

(*matrix1,matrix2*)
Append *matrix2* appended to *matrix1* as columns.

(*matrix1;matrix2*)
Append *matrix2* to *matrix1* as rows.

**avgRC(***expr,var* [*,h*]**)**
Return forward-difference quotient (average rate of change) of *expr* with respect to *var*. *h* is the step value; if omitted, *h*=0.001

**▸Bin**
*integer1* ▸Bin
Convert *integer1* to binary. Non-decimal integers must be preceeded with 0b or 0h. Arguments larger than 32-bit signed values are reduced with symmetric modulo operation.

**BldData** [*datavar]*
Create the data variable *datavar* based on the current graph settings. If *datavar* is omitted, the system variable *sysdata* is used. The variable is a table of function values evaluated at each plot point.

**ceiling()**
(*expr*), (*list*), (*matrix*)
Return the nearest integer that is greater than or equal to *expr* or each element of *list* or *matrix.* Argument may be real or complex.

**cFactor()**
*(expr[,var])*

*(list[,var]), (matrix[,var])*
Return complex factors of argument over a common denominator. If *var* is omitted, argument is factored with respect to all variables. If *var* is used, cfactor() tries to factor the argument toward factors which are linear in *var*.

**char(***integer***)**
Return string of character indicated by *integer*. *Integer* must be in the range 0 to 255.

**Circle** *x*, *y,r [,drawmode]*
Draw a circle on the Graph with radius *r* and center (*x*, *y*). All arguments are in window coordinate units.
*drawmode* = 1: draw circle (default)
*drawmode* = 0: turn off the circle
*drawmode* = -1: invert circle pixels

**ClrDraw**
Clear the Graph screen; reset Smart Graph

**ClrErr**
Clear error status and internal error context variables. Sets *errnum* to zero.

**ClrGraph**
Clears graphed functions or expressions.

**ClrHome**
Clears the home screen, and sets arbitrary constant suffixes to 1.

**ClrIO**
Clears the program I/O screen.

**ClrTable**
Clears table settings which apply to the *ASK* setting in the *Table Dialog* setup box.

**colDim(***matrix***)**
Return number of rows of *matrix*.

**colNorm(***matrix***)**
Return maximum of sums of absolute values of *matrix* column elements.

**comDenom()**
(*expr [,var]*)
(*list [,var]*)
(*matrix [,var]*)
Return reduced ratio of expanded numerator over expanded denominator, with respect to *var* if used. Using *var* can save time, memory and screen space, and result in expressions on which further operations are less likely to result in *Memory* errors.

**conj()**
(*expr*), (*list*), (*matrix*)
Return the complex conjugate of the argument. Undefined variables are treated as real.

**CopyVar**
*var1, var2*
Copy the contents of *var1* to *var2*. Unlike the *store* operation (→), *CopyVar* does not simplify the source variable. *CopyVar* must be used with non-algebraic variable types such as Pic and GDB.

**cos()**
(*expr*), (*list*)
Return the cosine of the argument. You may override the current angle mode with ° or ʳ.

(*matrix*)
Return the matrix cosine of square diagonalizable *matrix*, which is not the cosine of each element. Symbolic elements must have assigned values. The matrix cosine is calculated with floating-point arithmetic (Approx mode).

**cos$^{-1}$()**
(*expr*), (*list*)
Return the angle whose cosine is *expression*.

(*matrix*)
Return the matrix inverse cosine of square diagonalizable *matrix*, which is not the inverse cosine of each element.Results are found with floating-point arithmetic.

**cosh()**
(*expr*), (*list*)
Return the hyperbolic cosine of the argument.

(*matrix*)
Return the hyperbolic cosine of square diagonalizable *matrix*, which is not cosh() of each element. Results are found with floating-point arithmetic.

**cosh$^{-1}$()**
(*expr*), (*list*)
Return the inverse hyperbolic cosine of the argument.

(*matrix*)
Return the inverse hyperbolic cosine of square diagonalizable *matrix*, which is not $\cosh^{-1}$() of each element. Results are found with floating-point arithmetic.

**crossP()**
(*list1*,*list2*), (*matrix1*,*matrix2*)
Return the cross product of the arguments. Lists must have equal dimensions of 2 or 3. Matrices must both be either row or column vectors of equal dimensions 2 or 3.

**cSolve(***equation*,*var***)**
Return real and complex solutions for *var*. Complex results are possible in Real mode. Fractional powers with odd denominators use the principle branch, not the real branch. Specify complex variables with the underscore '_' suffix.

(*equation1* and *equation2* [and ...] ,{*var1*,*var2*[,...]}})
Return real and complex solutions for *var* to system of equations *equation1*, *equation2*, ... *var* may be a variable or a variable with a solution guess such as x=1+3i. Not all equation variables need be in the solution variable list. A close complex solution guess may be needed to get a solution.

**CubicReg**
*xlist*,*ylist*[,[*freqList*],[*catlLst*],[*catincList*]]
Calculate cubic polynomial regression and update all statistics variables. First four arguments must be variable names or c1-c99. Last argument need not be a variable name and cannot be c1-c99. All lists except *catincList* must have equal dimension. The regression equation is y = a*x^3 + b*x^2 + c*x + d.

**cumSum()**
(*list*), (*matrix*)
Return list of cumulative sum of elements or list, or return matrix of cumulative sums of columns from top to bottom.

**CustmOff**
Remove a custom toolbar.

**CustmOn**
Activate a custom toolbar as set up by Custom ... EndCustom block.

**Custom**
Set up a custom toolbar. Similar to ToolBar except that Title and Item statements cannot have labels.

**Cycle**
Transfer program control immediately to the next iteration of the current For, While or Loop loop.

**CyclePic**
*picString*, *n* [, [*wait*], *cycles*], [*direction*]]
Display each of *n* picture variables specified by *picString* for *wait* seconds. *cycles* specifies the number of display cycles. *direction* is set to 1 (default) for a circular cycle or to -1 for a forward-backward cycle. To display three variables *pic1, pic2* and *pic3*, *picString* is "pic" and *n* is 3.

**▸Cylind**
*vector* ▸Cylind
Display row or column *vector* in cylindrical form [r∠θ,z]. *vector* must be a row or column vector with three elements.

**cZeros(***expr*,*var***)**
Equivalent to exp▸list(cSolve(*expr*=0,*var*),*var*). Return list of real and complex solutions for *var* which make *expr*=0. Use the underscore '_' suffix to specify complex variables. *var* may be a variable name or a solution guess such as x = 1 + 3i.

({*expr1*,*expr2* [,...]},{*var1*,*var2* [,...]})
Return matrix of solutions for *var1, var2, ...* where the expressions are simultaneously zero. *var1, var2, ...* may be variable names or solution guesses such as x = 1 + 3i. Solution zeros may include real and complex solutions. Each solution matrix row represents a zero, with the components in the same order as the variable list. You need not include all the expression variables in the variable list. You may also include variables which are not in the expressions. Complex guesses are often necessary, and the guess may have to be close to the solution.

***d*****(***expr*,*var* [,*order*]**)**
(*list*,*var* [,*order*])
(*matrix*,*var* [,*order*])
Return derivative of *order* of *expr*, *list* or *matrix* argument with respect to *var*. *order* must be an

integer. If *order* is less than zero, the anti-derivative is returned. *d*() does not fully simplify the *expression* and *var* before finding the derivative.

**▸DD**
*number*▸DD
*list*▸DD
*matrix*▸DD
Return decimal equivalent of the angle argument. The current Angle Mode sets the argument as degrees or radians. The argument may be radians.

**▸Dec**
*number*▸Dec
Convert *number* to a decimal number, regardless of the current Base mode.

**Define**
Define *functionName*(*arg1*,*arg2*, ...)=*expr*
Create function *functionName()* with arguments *arg1, arg2. fName()* evaluates *expression* with the supplied arguments and returns the result. Do not use *arg1* or *arg2* as arguments when calling *functionName()*. This form of Define is equivalent to *expr→functionName*(*arg1*,*arg2*).

Define *functionName*(*arg1*,*arg2*,...)=Func
 *block*
EndFunc
Same as above except that *block* can include multiple expressions and use Return to return the result.

Define *programName*(*arg1*,*arg2*,...)=Prgm
 *block*
EndPrgm
Same as above except that program *programName()* is defined. Programs cannot return results.

**DelFold** *folder1* [,*folder2*, *folder3*, ...]
Delete folders. The *Main* folder cannot be deleted. An error message is shown if any folder contains variables.

**DelVar** *var1 [var2, var3, ...]*
Delete variables from memory.

**deSolve(***ode12*,*indVar*,*depVar***)**
Return general solution of 1st- or 2nd order ordinary differential equation *ode12* with independent variable *indVar* and dependent variable *depVar*. The prime symbol ' indicates the 1st derivative, two prime symbols ' ' indicate the second derivative.

(*ode1* and *inCond*,*indVar*,*depVar*)
Return particular solution of 1st-order differential equation *ode1* with initial condition *inCond*. *inCond* is in the form *depVar*(*inIndVar) = inDepVar;* for example, y(0) = 1. *inIndVar* and *inDepVar* can be variables with no stored values.

(*ode2* and *inCond1* and *inCond2*,*indVar*,*depVar*)
Return particular solution of 2nd-order differential equation *ode2* with initial conditions *inCond1* and *inCond2*. *inCond1* specifies the value of the independent variable at a point, in the form *depVar(inIndVal) = inDepVal*. *inCond2* specifies the value of the first derivative at a point in the form *depVar'*(*inIndVal*) = *in1stDeriv*.

(*ode2* and *BndCnd1* and *BndCnd2*,*indVar*, *depVar*)
Return particular solution of 2nd-order differential equation with boundary conditions *BncCnd1* and *BndCnd2*.

**det(***matrix* [,*tol*]**)**
Return determinant of square *matrix*. Any element less than *tol* is considered zero. The default *tol* is 5E-14 * max(dim(*matrix*)) * rowNorm(*matrix*).

**diag()**
(*list*), (*rowMatrix*), (*colMatrix*)
Return matrix with main diagonal elements of argument.

(*matrix*)
Return row matrix whose elements are the main diagonal elements of square *matrix*.

**Dialog**
Dialog :  *block* : EndDlog
Display a dialog box during program execution. *block* consists of Text, Request, DropDown and Title commands. Dialog box variables are displayed as default values. If [ENTER] is pressed, the variables are updated and the system variable *ok* is set to 1. If [ESC] is pressed, the variables are not updated and *ok* is set to zero.

**dim(*list*)**
Return dimension of list.

(*matrix*)
Return list of dimensions of *matrix* as {rows,columns}

(*string*)
Return dimension (number of characters) of *string*.

**Disp** [*expOrString1*], [*expOrString2*], ...
Display the program I/O screen. If any arguments (expressions or strings) are used, they are displayed on separate screen lines. Arguments are displayed in Pretty Print if Pretty Print mode is On.

**DispG**
Display the Graph Screen.

**DispHome**
Display the Home screen.

**DispTbl**
Display the Table screen. Use the cursor pad to scroll. Press [ENTER] or [ESC] to resume program operation.

**▸DMS**
*expr* ▸DMS, *list* ▸DMS, *matrix* ▸DMS
Display the argument as an angle in format DDDDDD°MM'SS.ss". ▸DMS converts from radians in radian mode.

**dotP()**
(*list1*,*list2*), (*vector1*,*vector2*)
Return the dot product of two lists or vectors. Vectors must both be row or column vectors.

**DrawFunc** *expr*
Draw *expr* as a function of *x* on the Graph screen.

**DrawInv** *expr*
Draw the inverse of *expr* on the Graph screen by plotting *x* values on the *y* axis and vice versa.

**DrawParm** *expr1, expr2* [,*tmin*] [,*tmax*] [,*tstep*]
Draw parametric expressions *expr1* and *expr2* with *t* as the independent variable, from *tmin* to *tmax* with *tstep* between each evaluated *t* value. The current Window variables are the defaults for *tmin, tmax* and *tstep*. Using the *t*-arguments does not change the Window settings. The *t*-arguments must be used if the current Graph mode is not Parametric.

**DrawPol** *expr,* [,θ*min*] [,θ*max*] [,θ*step*]
Draw *expr* as a polar graph with independent variable θ. Defaults for θ*min*, θ*max* and ,θ*step* are the current Window settings. Specifying θ-arguments does not change the Window settings. Theθ-arguments must be used if the Graph mode is not Polar.

**DrawSlp** *x1*, *y1*, *slope*
Display the Graph screen and draw the line y = *slope**(x-*x1*) + *y1*.

**DropDown** *title*, {*item1*, *item2*, ...},*var*
Display a drop-down menu with the name *title* in a Dialog box. The drop-down menu choices are 1:*item2*, 2:*item2* and so on. The number of the selected item is stored in *var*. If *var* exists and its value is in the range of items, the referenced item is the default menu selection.

**DrwCtour**
*expr*
*list*
Draw contours on the current 3D graph, in addition to the contours specfied by the *ncontour* system variable. *expr* or *list* specifies the z-values at which the contours are drawn. 3D graph mode must be set. *DrwCtour* sets the graph style to CONTOUR LEVELS.

**E (enter exponent)**
*mantissa*E*exponent*
Enter a number in scientific notation as *mantissa* * $10^{exponent}$. To avoid a decimal value result, use $10^{integer}$, instead.

**e^()**
(*expr*)
Raise *e* to the *expr* power. *e* is the natural logarithm base 2.718..., not the character 'e'. In Radian angle mode, you may enter complex numbers in the polar format $re^{i\theta}$.

(*list*)
Return list of *e* raised to each element in *list*.

(*matrix*)
Return the matrix exponential of square *matrix*, which is not the same as *e* raised to the power of each element. See cos() for calculation

details. *matrix* must be diagonalizable. The result always contains floating-point numbers.

**eigVc(***matrix***)**
Return matrix of eigenvectors of square *matrix*, whose elements may be real or complex. The eigenvectors are not unique, but are normalized such that if the eigenvector is

$$V=\left[x_1,x_2,\ldots,x_n\right]$$

then

$$\sqrt{x_1^2+x_2^2+\ldots x_n^2}\ =1$$

**eigVl(***matrix***)**
Return the eigenvalues of square diagnonalizable *matrix*, whose elements may be real or complex.

**Else** See If
**ElseIf** See If

**EndCustm** See Custom
**EndDlog** See Dialog
**EndFor** See For
**EndFunc** See Func
**Endif** See If
**EndLoop** See Loop
**EndPrgm** See Prgm
**EndTBar** See ToolBar
**EndTry** See Try
**EndWhile** See While

**entry(**[*integer*]**)**
Return previous entry-line expression from the history area. If used, *integer* cannot be an expression and must be in the range 1 to 99. See also ans().

**exact()**
(*expr* [,*tol*]), (*list* [,*tol*]), (*matrix* [,*tol*])
Evaluate argument with Exact mode arithmetic regardless of the current Mode setting. *tol*, if used, specifies the conversion tolerance and is zero by default.

**Exec** *string* [,*expr1*] [,*expr2*] ...
Execute *string* interpreted as Motorola 68000 assembly language op-codes. *expr1*and *expr2* are optional input arguments. Misinformed use of *Exec* can lock up the calculator and cause data loss.

**Exit**
Exit the current For, While or Loop block.

**exp ▸list(***expr*,*var***)**
Convert *expr* to a list of the right-hand sides of equations separated by 'or' in *expr*. Used to extract individual solutions from results of solve(), cSolve(), fMin() and fMax().

**expand()**
(*expr*), (*list*), (*matrix*)
Expand argument with respect to all variables with transformation into a sum and/or difference of simple terms. See also Factor()

(*expr,var*), (*list*,*var*), (*matrix*,*var*)
Expand argument with respect to *var* by collecting similar powers of *var* and sorting the factors with *var* as the main variable. expand() also distributes logarithms and fractional powers regardless of *var*. Specifying *var* can save time, memory and result in a smaller expression. If the argument only contains one variable, specifying it as *var* may result in more complete partial factorization. propFrac() is faster but less extreme than expand(). See also comDenom() and tExpand().

**expr(***string***)**
Return the evaluated expression in *string*.

**ExpReg**
*xlist*,*ylist*[,[*freqList*],[*catlLst*],[*catincList*]]
Calculate exponential polynomial regression and update all statistics variables. First four arguments must be variable names or c1-c99. Last argument need not be a variable name and cannot be c1-c99. All lists except *catincList* must have equal dimension. The regression equation is y = a*b^x.

**factor()**
(*expr*), (*list*), (*matrix*)
Factor argument with respect to all variables over a common denominator. Argument is factored as much as possible toward linear rational factors without introducing new non-real sub-expressions.

(*expr*,*var*), (*list*,*var*), (*matrix*,*var*)
Factor argument with respect to *var*, as much as possible toward real factors linear in *var*, even if irrational constants or subexpressions are

introduced. Also, comDenom() can achieve partial factoring if factor() is too slow or exhausts memory. See also cFactor()

(*ratNum*)
Return factors of rational number *ratNum*. Use isPrime(), instead, to more quickly determine if *ratNum* is prime.

**Fill**
*expr,matVar*
*expr,listVar*
Replace each element in *matVar* or *listVar* with *expr. matVar* and *listVar* are variable names, and the variables must exist.

**floor()**
(*expr*), (*list*), (*matrix*)
Return greatest integer which is less than or equal to the argument. The argument may be real or complex. floor() is identical to int(). See also ceiling().

**fMax(***expr*,*var***)**
Return Boolean expression specifying possible values of *var* which maximize *expr* with respect to *var*, or locate the least upper bound. Use the "with" operator "|" to limit the solution range or add constraints. In Approx mode, fMax() finds one approximate local maximum.

**fMin(***expr*,*var***)**
Return Boolean expression specifying possible values of *var* which minimize *expr* with respect to *var*, or locate the greatest lower bound. Use the "with" operator "|" to limit the solution range or add constraints. In Approx mode, fMin() finds one approximate local minimum.

**FnOff** [1] [,2] ... [,99]
Deselects all Y= functions, with no arguments. Optional arguments specify Y= functions to deselect.

**FnOn** [1] [,2] ... [,99]
Selects all Y= functions, with no arguments. Optional arguments specify Y= functions to select. In 3D mode only, selecting any function deselects all other functions.

**For**
For *var*,*low*,*high* [,*step*] :  *block* : EndFor
Execute each statement in *block* iteratively for each value of *var*. *var* ranges from *low* to *high* and increments by *step. step* may be positive or negative and the default is 1. *var* cannot be a system variable.

**format(***expr*,*format***)**
Return expression *expression* as a string formatted with the numeric format string *format*:

"F[*n*]" Fixed format; *n*  is number of digits after the decimal point.

"S[*n*]" Scientific format; *n* is the number of digits after the decimal point.

"E[*n*]" Engineering format; *n* is the number of digits after the first significant digit. Mantissa radix point is adjusted so that exponent is a power of three.

"G[*n*][*c*]" General format; samed as fixed, but digits to the left of the radix are separated in groups of three, separated by the *c* character. By default *c* is a comma; if *c* is a period, the radix point is a comma.

The suffix [R*c*] may be added to any of the *format* codes. *c* is a single character which specifies the radix point character.

**fpart()**
(*expr*), (*list*), (*matrix*)
Return the fractional part of the argument, which may be real or complex.

**Func**
Func :  *block* : EndFunc
*Func* is required as the first statement of a multi-statement function definition.

**gcd()**
(*number1*,*number2*), (*list1*,*list2*), (*matrix1*,*matrix2*)
Return the greatest common denominator of the arguments. The GCD of two fractions is the GCD of the numerators divided by the least common multiple of the denominators. The GCD of fractional floating-point numbers is 1.0 in Auto or Approx mode. If the arguments are lists or matricec, the GCD of each corresponding element is returned.

**Get** *var*
Retrieve a CBL or CBR value from the link port and store it in *var*.

**GetCalc** *var*
Retrieve a value from the link port from another calculator and store it in *var*.

**GetConfg**
Return a list of calculator attribute pairs. The first pair element is a string which specifies the attribute, and the second pair element is the attribute. The attribute names are the same for the TI-89 and the TI-92+, but the attributes may differ. The "Cert. Rev. #" attribute pair appears in the list only if a certificate has been installed.

{
"Product Name",*productName*,
"Version","*versionString*",
"Product ID",*idString*,
"ID#",*id#String*,
"Cert. Rev. #",*certRev#*,
"Screen Width",*screenWidth*,
"Screen Height",*screenHeight*,
"Window Width",*windowWidth*,
"Window Height",*windowHeight*,
"RAM size",*ramSize*,
"Free RAM",*freeRAM*,
"Archive Size",*archiveSize*,
"Free Archive",*freeArchive*
}

**getDenom(***expr***)**
Return the reduced common denominator of *expression*.

**getFold()**
Return a string which is the current folder name.

**getKey()**
Return key code of a pressed key as an integer, or return zero if no key is pressed.

**getMode(***modeString***)**
Return a string which is the current setting for *modeString*.

("ALL")
Return a list of string pairs of all Mode settings. The first element of the string pair is the mode name string; the second element is the mode setting string. See SetMode() for possible settings.

{
"Graph", *graphType*,
"Display Digits", *digitsFormat*,
"Angle", *angleUnits*,
"Exponential Format", *expFormat*,
"Complex Format", *complexFormat*,
"Vector Format", *vectorFormat*,
"Pretty Print", *prettyPrintStatus*,
"Split Screen", *splitScreenMode*,
"Split 1 App", *app1Name*,
"Split 2 App", *app2Name*,
"Number of Graphs", *numGraphs*,
"Graph 2", *graphType*,
"Split Screen Ratio", *ratio*,
"Exact/Approx", *exactApproxMode*,
"Base", *numberBase*
}

**getNum(***expr***)**
Return numerator of *expression* reduced to a common denominator.

**getType(***varName***)**
Return string indicating the data type of variable *varName*.

"ASM" assembly-language program
"DATA" Data type
"EXPR" expression; includes complex, arbitrary, undefined, ∞, -∞, TRUE, FALSE, π, *e*
"FUNC" Function
"GDB" Graph data base
"LIST" List
"MAT" Matrix
"NONE" Variable does not exist
"NUM" Real number
"OTHER" Reserved for future use
"PIC" Picture
"PRGM" Program
"STR" String
"TEXT" Text type
"VAR" Name of another variable

**getUnits()**
Return a list of strings which specify the default units. Constants, temperature, amount of a substance, luminous intensity and acceleration are not included. The list has the form {"*system*","*cat1*","*unit1*","*cat2*","*unit2*",...} where *system* is the unit system: SI, ENG/US or CUSTOM. The *cat* strings specify the category, and the *unit* strings specify the corresponding default units.

**Goto** *label*
Transfer program control to *label*.

**Graph**
*expr* [*,var*] (function graph)
*xExpr,yExpr* [,*var*] (parametric graph)
*expr* [,θvar] (polar graph)
*expr* [*,xvar*] [*,yvar*] (3D graph)
Graph the *expr* argument with the current Graph mode. Expressions created with Graph (or Table) are assigned increasing function numbers starting with 1. Modify or delete them with the [F4] Header function in the table display. The currently selected Y= functions are not graphed. The independent variable of the current graph mode is used if a *var* argument is omitted. Use ClrGraph to clear the functions, or start the Y= editor to enable the system variables.

**▸Hex**
*integer1*▸Hex
Convert *integer1* to hexadecimal. Non-decimal integers must be preceeded with 0b or 0h. Arguments larger than 32-bit signed values are reduced with symmetric modulo operation.

**identity(***expr***)**
Return identitiy matrix with dimension of *expr*.

**If**
If *BooleanExpr* :  *statement*
Execute single *statement* if *BooleanExpr* evaluates to TRUE.

If *BooleanExpr* then :  *block* :EndIf
Execute *block* if *BooleanExpr* evaluates to TRUE.

If *BooleanExpr* then :  *block1*
Else :  *block2* : EndIf
If *BooleanExpr* evaluates to TRUE, execute *block1* but not *block2*; otherwise execute *block2* but not *block1.*

If *BooleanExpr1* Then :  *block1*
ElseIf *BooleanExpr2* Then :  *block2*
...
ElseIf *BooleanExprN* Then :  *blockN*
EndIf
Execute *block1* only if *BooleanExpr1* evaluates to TRUE, execute *block2* only if *BooleanExpr2* evaluates to TRUE, etc.

**imag()**
(*expr*), (*list*), (*matrix*)
Return the imaginary part of the argument. All undefined variables are treated as real variables.

**Input** [[*promptString*,]*var*]
If no arguments, pause program execution, display the Graph screen and update *xc* and *yc* (or *rc* and θ*c* in Polar mode) by positioning the cursor.

If argument *var* is used, the program pauses, displays *promptString* on the Program I/O screen and waits for the entry of an expression. The expression is stored in *var*. If *promptString* is omitted, "?" is displayed for the prompt.

**InputStr** [*promptString*,]*var*
Pause program execution, display *promptString* on the Program I/O screen and wait for the entry of an expression, which is stored as a string in *var*. "?" is displayed as a prompt if *promptStrin* is omitted.

**inString(***sourceString*,*targetString*[,*start*]**)**
Return position at which *targetString* starts in *sourceString*. The search for *targetString* begins at character number *start*, if *start* is used. *start* is 1 by default.

**int()**
(*expr*), (*list*), (*matrix*)
Return the greatest integer that is less than or equal to the argument, which may be real or complex. int() is identical to floor().

**intDiv()**
(*number1*,*number2*), (*list1*,*list2*), (*matrix1*,*matrix2*)
Return the signed integer part of the first argument divided by the second argument.

**integrate**
See ∫().

**iPart()**
(*number*), (*list*), (*matrix*)
Return the integer part of the argument, which may be real or complex.

**isPrime(***number***)**
Return True or False to indicate if *number* is prime. Display error message if *number* has more than about 306 digits and no factors less than or equal to 1021.

**Item**
*itemNameString*
*itemNameString,label*
Set up a drop-down menu within Custom ... EndCustm, or ToolBar ... EndTBar blocks. In a Custom block, specifies the text that is pasted. In a ToolBar block, specifies the branch label.

**Lbl** *labelName*
Define a label with name *labelName* in a program. Goto *labelName* transfers control to the instruction following *labelName*. *labelName* must meet same requirements as variable names.

**lcm()**
(*number1,number2*)
(*list1,list2*)
(*matrix1*,*matrix2*)
Return the least common multiple of the two arguments. The LCM of two fractions is the LCM of the numerators divided by the GCM of the denominators. The LCM of fractional floating point numbers is their product.

**left()**
(*sourceString*[,*num*])
Return the leftmost *num* characters from *sourceString*.

(*list*[,*num*])
Return the leftmost *num* elements of *list*.

(*comparison*)
Return the left-hand side of the equality or inequality of *comparison*.

**limit()**
(*expr*,*var*,*point*[,*direction*])
(*list*,*var*,*point*[,*direction*])
(*matrix*,*var*,*point*[,*direction*])
Return the limit of the argument *expr, list*, or *matrix* with respect to *var* at *point*. The limit is taken from the left if *direction* is negative, from the right if *direction* is positive, or from both directions otherwise. The default *direction* is both.

**Line** *xStart*,*yStart*,*xEnd*,*yEnd*[,*mode*]
Display Graph screen and draw a line, including the endpoints, from (*xStart*,*yStart*) to (*yStart*, *yEnd*) according to *mode*:
*mode*=1: draw the line (default)
*mode*=0: turn off the line
*mode*=-1: invert the line

**LineHoriz** *y*[,*mode*]
Display Graph screen and draw a horizontal line at window coordinate *y*, according to *mode*:
*mode*=1: draw the line (default)
*mode*=0: turn off the line
*mode*=-1: invert the line

**LineTan** *expr1*,*expr2*
Display Graph screen and draw line tangent to *expr1* at point x = *expr2*. The independent variable for *expr1* is x.

**LineVert** *x*[,*mode*]
Display Graph screen and draw a vertical line at window coordinate *x*, according to *mode*:
*mode*=1: draw the line (default)
*mode*=0: turn off the line
*mode*=-1: invert the line

**LinReg** *xlist*,*ylist*[,[*freqList*],[*catlLst*],[*catincList*]]
Calculate linear regression and update all statistics variables. First four arguments must be variable names or c1-c99. Last argument need not be a variable name and cannot be c1-c99. All lists except *catincList* must have equal dimension. The regression equation is y = a*x + b.

**list▸mat(***list*[,*rowElements*]**)**
Create a matrix whose elements are filled row-by-row from. *rowElements* specifies the number of elements in each matrix row, if used. The default is one matrix row. If *list* does not fill the matrix, zeros are added.

**Δlist(***list***)**
Return a list which is the difference between successive elements of *list*. Each element of *list* is subtracted from the next element, so the returned list is always one element shorter than *list*.

**ln()**
(*expr*), (*list*)
Return the natural logarithm of the argument.

(*matrix*)
Return the matrix natural logarithm of square diagonalizable *matrix*. This is not the same as finding the logarithm of each matrix element.

Floating-point results are used. See cos() for calculation details.

**LnReg** *xlist*,*ylist*[,[*freqList*],[*catlLst*],[*catincList*]]
Calculate logarithmic regression and update all statistics variables. First four arguments must be variable names or c1-c99. Last argument need not be a variable name and cannot be c1-c99. All lists except *catincList* must have equal dimension. The regression equation is y = a + b*ln(x).

**Local** *var1* [,*var2*, *var3*, ...]
Declare the arguments as local variables in a program or function. Local variables exist only during program or function execution. Local variables must be used for loop index variables and for storage in multi-line functions, since global variables are not allowed in functions.

**Lock** *var1* [,*var2*, *var3*, ...]
Lock the argument variables. Locked variables cannot be changed or deleted unless Unlock() is used.

**log()**
(*expr*), (*list*)
Return the base-10 logarithm of the argument.

(*matrix*)
Return the matrix base-10 logarithm of square diagonalizable *matrix*. This is not the same as finding the logarithm of each *matrix* element. Floating-point arithmetic is used. Refer to cos() for calculation details.

**Logistic** *xlist*,*ylist*[,[*freqList*],[*catlLst*],[*catincList*]]
Calculate logistic regression and update all statistics variables. First four arguments must be variable names or c1-c99. Last argument need not be a variable name and cannot be c1-c99. All lists except *catincList* must have equal dimension. The regression equation is $y = \frac{a}{1+b\cdot e^{c\cdot x}} + d$ where *e* is the natural logarithm base.

**Loop**
Loop :  *block* : EndLoop
Repeatedly execute *block* until a Goto or Exit instruction is executed.

**LU** *matrix*, *lMat*, *uMat*, *pMat* [,*tol*]
Calculator the Doolittle LU (lower-upper) matrx decomposition or real or complex *matrix*. *matrix* must be a matrix name. The results are
*lMat* = lower triangular matrix
*uMat* = upper triangular matrix
*pMat* = permutation matrix
such that *lMat* * *uMat* = *pMat* * *matrix*
Any matrix element is set to zero if its value is less than *tol* and *matrix* contains no floating-point or symbolic elements. The default value for *tol* is 5E-14 * max(dim(*matrix*)) * rowNorm(*matrix*). Computations are done with floating-point arithmetic in Approx mode.

**mat▸list(***matrix***)**
Return a list whose elements are those of *matrix*, row by row.

**max()**
(*expr1*,*expr2*), (*list1*,*list2*), (*matrix1*,*matrix2*)
Return the maximum of the arguments. List and matrix arguments are compared element by element.

(*list*)
Return the maximum element of *list*.

(*matrix*)
Return a row vector whose elements are the maximum elements of the columns of *matrix*.

**mean(***list*[,*freqList*]**)**
Return the mean of the elements of *list*. *freqList* indicates the number of occurrences of the corresponding *list* element.

(*matrix*[,*freqMatrix*])
Return a row vector whose elements are the means of the *matrix* columns. *freqMatrix* indicates the number of occurrences of the corresponding *matrix* element.

**median()**
(*list*)
Return the median of the elements of *list*. All *list* elements must simplify to numbers.

(*matrix*)
Return a row vector whose elements are the medians of the columns of *matrix*. All *matrix* elements must simplify to numbers.

**MedMed** *xlist*,*ylist*[,[*freqList*],[*catlLst*],[*catincList*]]
Calculate median-median regression and update all statistics variables. First four arguments must be variable names or c1-c99. Last argument need not be a variable name and cannot be c1-c99. All lists except *catincList* must have equal dimension. The regression equation is y = a*x + b.

**mid()**
For both variations of mid(): if *count* is omitted or greater than the dimension of the argument, the complete argument is returned. *count* must be greater than or equal to zero. If *count* = 0, an empty string or list is returned.

(*string*,*start*[,*count*])
Return *count* characters from *string* beginning at character position *start.*

(*list*,*start*[,*count*])
Return *count* elements of *string* beginning at element *start.*

**min()**
(*expr1*,*expr2*), (*list1*,*list2*), (*matrix1*,*matrix2*)
Return the minimum of the arguments. List and matrix arguments are compared element by element.

(*list*)
Return the minimum element of *list*.

(*matrix*)
Return a row vector whose elements are the minimum elements of the columns of *matrix*.

**mod()**
(*expr1*,*expr2*), (*list1*,*list2*), (*matrix1*,*matrix2*)
Return the first argument modulo the second argument where
mod(x,0) = x
mod(x,y) = x - y*floor(x/y)

The result is periodic in the second argument when that argument is non-zero. The result is either zero or has the same sign as the second argument.

**MoveVar** *var*,*curFolder*,*newFolder*
Move *var* from *curFolder* to *newFolder.* *newFolder* is created if it does not exist.

**mRow(***expr*,*matrix*,*index***)**
Return *matrix* with row *index* of *matrix* multiplied by *expr*.

**mRowAdd(***expr*,*matrix*,*index1*,*index2***)**
Return *matrix* with row *index2* replaced with *expr* * row *index1* + row *index2*

**nCr(***expr1*,*expr2***)**
Return number of combinations of *expr1* items taken *expr2* at a time, for *expr1* ≥ *expr2* ≥ 0, and integer *expr1* and *expr2.* This is the binomial coefficient.

(*expr*,*n*)
If *n* is zero or a negative integer, return 0.

If *n* is a positive integer or non-integer, return

$$\frac{\mathrm{expr}!}{n!(\mathrm{expr}-n)!}$$

(*list1*,*list2*)
(*matrix1*,*matrix2*)
Return list of nCr() of corresponding element pairs of each argument. Both arguments must have the same dimension.

**nDeriv()**
(*expr*,*var*[,*h*])
Return an estimate of the numerical derivative of *expr* with respect to *var* using the central difference quotient formula. *h* is the step size and defaults to 0.001.

(*expr*,*var*,*list*)
Return list of derivative estimates for each step size in *list*.

(*list*,*var*[,*h*])
(*matrix*,*var*[,*h*])
Return list or matrix of derivative estimates for each element of *list* or *matrix*.

**NewData** *var*,*list1*[,*list2*] [,*list3*] ...
Create data variable *var* whose columns are the *lists* in order. The *list*s can be lists, expressions or list variable names. Make the new variable current in the Data/Matix editor.

*var*,*matrix*
Create data variable *var* based on *matrix*.

*sysdata*,*matrix*
Load *matrix* in the *sysdata* system data variable.

**NewFold** *folderName*
Create a folder with the name *folderName*, and make that folder current.

**newList(***expr***)**
Return a list of zeros with dimension *expr*.

**newMat(***exprRow*,*exprCol***)**
Return a matrix of zeros with *exprRow* rows and *exprCol* columns.

**NewPic** *matrix*,*var* [,*maxRow*] [,*maxCol*]
Create a pic variable *var* based on *matrix*. *matrix* has two columns, and each row is the coordinates of a pixel which is turned 'on' in the picture. *var* is replaced if it exists. The default size of *var* is the minimum boundary area required by the *matrix* coordinates. *maxRow* and *maxCol* specify optional maximum boundary limits for *var*.

**NewPlot** *n*,*type*,*xlist*[,[*ylist*],[*frqList*],[*catList*], [*incList*], [*mark*],[*bucketSize*]]
Create a new data plot definition for plot number *n*, where *n* is 1 to 9. *xlist* and *ylist* are the data to plot. *frqList* is the frequency list, *catList* is the category list and *incList* is the category include list. *type* specifies the plot type:
1 = scatter plot
2 = xyline plot
3 = box plot
4 = histogram
5 = modified box plot

*mark* specifies the plot symbol:
1 = box
2 = cross
3 = plus
4 = square
5 = dot

*bucketSize* must be > 0 and specifies the historgram bucket width and varies with *xmin* and *xmax*. *bucketSize* defaults to 1. *incList* need not be a variable name and cannot be c1 - c99. The other list arguments must be variable names or c1 - c99.

**NewProb**
Clear all unlocked and unarchived single-character variable names in the current folder. Turn off all function and stat plots in the current graph mode. Perform ClrDraw, ClrErr, ClrGraph, SlrHome, ClrIO and ClrTable.

**nInt(***expr*,*var*,*lower*,*upper***)**
Return approximate integral of *expr* with respect to *var* with integration limits *lower* and *upper*. *var* must be the only variable in *expr*. *lower* and *upper* must be constants, +∞ or -∞. The adaptive algorithm uses weighted sample values of the integrand in the interval *lower*<*var*<*upper*. nInt() attempts an accuracy of six significant digits and displays the warning message "Questionable accuracy" if the accuracy may be less. Nested nInt() calls can be used for multiple integration, and the integration limits can depend on integration variables outside them.

**norm(***matrix***)**
Return the Forbenius norm of *matrix*.

**not**
not *BooleanExp*
not *integer*
not *list*
not *matrix*
For Boolean arguments, return true, false or a simplified Boolean expression. For integer arguments, return the 1's complement of *integer*. Results are displayed according to the Base mode. Use the prefixes 0b or 0h for binary and hexadecimal integers. If the integer argument is larger than the maximum signed 32-bit binary number, a symmetric modulo operation brings the argument into range. Auto or Exact mode must be used with integer arguments.

**nPr(***expr1*,*expr2***)**
For integer *expr1* and *expr2*, and *expr1* ≥ *expr2* ≥ 0, return the number of permutations of *expr1* items taken *expr2* at a time.

(*m*,*r*)
If *r* = 0, return 1. Otherwise return m!/(m-r)!

**nSolve(***eqn*,*var***)**
Solve for eqn for one approximate solution of *var*. *var* may be a variable name in *eqn*, or a variable name with a solution guess in the form *var = guess.* Use the "|" operator to constrain the solution range. nSolve() will return the string "no solution found" if it cannot find a plausible solution.

**OneVar** *xList*[[,*freqList*][,*catList*][,*incList*]]
Calculate one-variable statistics and update system statistics variables. All lists must have equal dimension except *incList*. The first three arguments must be variable names of c1 - c99. *incList* need not be a variable name and cannot be c1 - c99.

**or**
*exp1* or *exp2*
*list1* or *list2*
*matrix 1* or *matrix 2*
Return *true* or *false* or a simplifed form.

*integer1* and *integer2*
Bit-by-bit 32-bit integer logical *or*. Arguments larger than 32-bit signed values are reduced with symmetric modulo operation. Mode must be Auto or Exact.

**ord()**
(*string*), (*list*)
Return numeric code of the first character in *string*. Return list of numeric codes of first characters of strings in *list*.

**Output** *row*,*column*,*exprOrString*
Display expression or string *exprOrString* at text coordinates *row*, *column*. *exprOrString* can include conversion operations such as ▸DD. *exprOrString* is pretty-printed if Pretty Print is on.

**P▸Rx()**
(*rExpr*,*θExpr*), (*rList*,*θList*), (*rMatrix*,*θMatrix*)
Return the x-coordinate of the (r,*θ*) pair*θ* is interpreted in degrees or radians according to the current angle mode. Use ° or $^{r}$ to over-ride the current mode setting.

**P▸Ry()**
(*rExpr*,*θExpr*), (*rList*,*θList*), (*rMatrix*,*θMatrix*)
Return the y-coordinate of the (r,*θ*) pair*θ* is interpreted in degrees or radians according to the current angle mode. Use ° or $^{r}$ to over-ride the current mode setting.

**part(***expr*[,*n*]**)**
Extract subexpressions of expression *expr*. *n* is a non-negative integer.

(*expr*)
Simplify *expr* and return the number of top-level arguments or operands. Return 0 if *expr* is a number, variable or symbolic constant.

(*expr*,0)
Simplify *expr* and return a string which contains the top-level function name or operator. Return string(*expr*) if *expr* is a number, variable or symbolic constant.

(*expr*,*n*)
Simplify *expr* and return the *n*th argument or operand, where *n* is greater than zero and less than or equal to the number of operands returned by part(*expr*). Otherwise return an error.

**PassErr**
Pass an error to the next level. Typically used in an Else clause (in place of ClrErr), when error handling is not yet determined.

**Pause** [*expr*]
Suspend program execution until [ENTER] is pressed. If included, *expr* is displayed on the Program I/O screen. If *expr* does not fit on the screen, the cursor pad scrolls the display. *expr* can include conversion operation suffixes such as ▸DD.

**PlotsOff** [1] [,2] [,3] ... [,9]
Turn all plots off with no arguments, or turn the specified plots off. Only affects the current graph in 2-graph mode.

**PlotsOn** [1] [,2] [,3] ... [,9]
Turn all plots on with no arguments, or turn the specified plots on. Only affects the current graph in 2-graph mode.

**▸Polar**
*vector* ▸Polar
Display vector in polar form [r∠θ]. *vector* must be a row or column vector of dimension 2. Can only be used at the end of an entry line, and does not update *ans*.

*complexValue* ▸Polar
Display *complexValue* in polar form: (r∠θ) in Degree mode, or $re^{i\theta}$ in Radian mode. *complexValue* can have any complex form, but $re^{i\theta}$ causes an error in Degree mode.

**polyEval()**
(*list*,*expr*), (*list*,*list2*)
Interpret *list* as the coefficients of a polynomial in descending degree, and return the polynomial evaluated for *expr* or each element of *list2*.

**PopUp** *itemList*,*var*
Display a pop-up menu with the character strings in *itemList*, wait for the press of a number key, and store the number in *var*. If *var* exists and contains a valid item number, that item is the default choice. *itemList* must have at least one character string.

**PowerReg**
*xlist*,*ylist*[,[*freqList*],[*catlLst*],[*catincList*]]
Calculate power regression and update all statistics variables. First four arguments must be variable names or c1-c99. Last argument need not be a variable name and cannot be c1-c99. All lists except *catincList* must have equal dimension. The regression equation is y = ax$^b$.

**Prgm**
Prgm : *block* : EndPrgm
Required instruction that identifies the first line of a TI-Basic program.

**product(***list* [,*start*[,*end*]]**)**
Return the product of the elements of *list*, from *start* to *end*.

(*matrix* ,[*start*[,*end*]])
Return row vector whose elements are the product of the column elements of *matrix*. *start* and *end* specify a row range.

**Prompt** *var1*[,*var2*][,*var3*]...
Display one prompt for each argument on the Program I/O screen. The prompt is the variable name suffixed with "?". The expression entered for each prompt is stored in the variable.

**propFrac(***rationalNumber***)**
Return *rationalNumber* as an integer and fraction with the same sign, where the fraction denominator is greater than the numerator.

(*rationalExpr*,*var*)
Return *rationalExpr* as a sum of proper ratios and polynomial in *var*. The degree of *var* in each ratio denominator is greater than the degree in the numerator. Similar powers of *var* are collected, and the terms and powers are sorted with respect to *var*.

(*rationalExpr*)
Return a proper fraction expansion with respect to the most main variable. The polynomial coefficients are then made proper with respect to their most main variable, and so on.

**PtChg** *x*,*y*
*xList*,*yList*
Display Graph screen and reverse pixel nearest to window coordinates (*x*,*y*)

**PtOff** *x*,*y*
*xList*,*yList*
Display Graph screen and turn off pixel nearest window coordinates (*x*,*y*).

**PtOn** *x*,*y*
*xList*,*yList*
Display Graph screen and turn on pixel nearest window coordinate (*x*,*y*).

**PtTest()**
(*x*,*y*), (*xList*,*yList*)
Return True if the pixel nearest window coordinates (*x*,*y*) is on, otherwise return False.

**PtText** *string*,*x*,*y*
Display Graph screen and place *string* with the upper-left corner of the first character at the pixel nearest window coordinates (*x*,*y*)

**PxlChg** *row*, *col*
*rowList*,*colList*
Display Graph screen and reverse the pixel at pixel coordinates (*row*,*col*) or (*rowList*,*colList*)

**PxlCrcl** *row*,*col*,*r* [,*drawMode*]
Display Graph screen and draw circle with radius *r* pixels and center at pixel coordinates (*row*,*col*).
If *drawMode* = 1, draw the circle (default).
If *drawMode* = 0, turn off the circle.
If *drawMode* = -1, invert pixels along the circle .

**PxlHorz** *row*[,*drawMode*]
Display Graph Screen and draw horizontal line at pixel coordinate *row*.
If *drawMode* = 1, draw the line (default).
If *drawMode* = 0, turn the line pixels off.
If *drawMode* = -1, invert the line pixels.

**PxlLine**
*rowStart*,*colStart*,*rowEnd*,*colEnd*[,*drawMode*]
Display Graph screen and draw line from pixel coordinates (*rowStart*,*colStart*) to (*rowEnd*, *colEnd*), including both endpoints.
If *drawMode* = 1, draw the line (default).
If *drawMode* = 0, turn the line pixels off.
If *drawMode* = -1, invert the line pixels.

**PxlOff**
*row*,*col*
*rowList*,*colList*
Display the Graph screen and turn off the pixel at pixel coordinates (*row*,*col*).

**PxlOn**
*row*,*col*
*rowList*,*colList*
Display Graph screen and turn on the pixel at pixel coordinates (*row*,*col*).

**PxlTest()**
(*row*,*col*), (*rowList*,*colList*)
Return True if the pixel at pixel coordinates (*row*,*col*) is on, otherwise return False.

**PxlText** *string*,*row*,*col*
Display Graph screen and place *string* with the upper-left corner of the first character at the pixel coordinates (*row*,*col*)

**PxlVert** *col*[,*drawMode*]
Display Graph Screen and draw vertical line at pixel coordinate *col*.
If *drawMode* = 1, draw the line (default).
If *drawMode* = 0, turn the line pixels off.
If *drawMode* = -1, invert the line pixels.

**QR** *matrix*,*qMatName*,*rMatName*[,*tol*]
Calculate Householder QR factorization of real or complex *matrix*. The Q and R result matrices are stored to the *matName* variables: Q is unitary, R is upper triangular. If the matrix has floating-point elements and no symbolic variables, then any matrix element less than *tol* is set to zero. The default value for *tol* is 5E-14 * max(dim(*matrix*)) * *rowNorm*(*matrix*). Computations are done with floating-point arithmetic if the mode is Approximate.

**QuadReg**
*xlist*,*ylist*[,[*freqList*],[*catlLst*],[*catincList*]]
Calculate quadratic polynomial regression and update all statistics variables. First four arguments must be variable names or c1-c99. Last argument need not be a variable name and cannot be c1-c99. All lists except *catincList* must have equal dimension. The regression equation is $y = ax^2 + bx + c$.

**QuartReg**
*xlist*,*ylist*[,[*freqList*],[*catlLst*],[*catincList*]]
Calculate quartic polynomial regression and update all statistics variables. First four arguments must be variable names or c1-c99. Last argument need not be a variable name and cannot be c1-c99. All lists except *catincList* must have equal dimension. The regression equation is $y = ax^4 + bx^3 + cx^2 + dx + e$.

**R▸Pθ()**
(*xExpr*,*yExpr*), (*xList*,*yList*)
(*xMatrix*,*yMatrix*)
Return the θ-coordinate of the (*x*,*y*) arguments, as either a radian or degree angle, depending on the current Angle mode.

**R▸Pr()**
(*xExpr*,*yExpr*), (*xList*,*yList*), (*xMatrix*,*yMatrix*)
Return the r-coordinate of the (*x*,*y*) arguments.

**rand(**[*n*]**)**
*n* is a non-zero integer. With no argument, return the next random number between 0 and 1 in the sequence. When *n*>0, return random integer in the interval [1,*n*]. When *n*<0, return random integer in the interval [-*n*,-1].

**randMat(***numRows*,*numColumns***)**
Return matrix with dimensions (*numRows*, *numColumns*) whose elements are random integers between -9 and 9.

**randNorm(***mean*,*sdev***)**
Return floating-point number from the standard distribution with *mean* and standard deviation *sdev*.

**randPoly(***var*,*order***)**
Return a polynomial of *order* in *var* whose coefficients are random integers from -9 to 9, where the leading coefficient is not zero.

**RandSeed** *n*
Reseed the random number generator. If *n* = 0, reseed with the factory defaults. Otherwise, *n* is

used to generate the two system variables *seed1* and *seed2*.

**RclGDB** *GDBvar*
Restore all Graph database settings from the variable *GDBvar*. See *StoGDB* for the settings.

**RclPic** *picVar*[,*row*,*column*]
Display Graph screen and add the picture *picVar* at the upper-left corner coordinates of (*row*,*column*), by logically OR-ing *picVar* with the Graph screen. The default coordinates are (0,0).

**real()**
(*expr*), (*list*), (*matrix*)
Return the real part of *expr*, or the real parts of the elements of *list* or *matrix*. Undefined variables are treated as real. See also *imag()*.

**▸Rect**
*vector* ▸Rect
Display *vector* in rectangular form [x,y,z]. *vector* must be a row or column vector with dimension 2 or 3. Rect can only be used at the end of an entry line and does not update *ans*.

*complexValue* ▸Rect
Display *complexValue* in rectangular for a + b*i*. *complexValue* may have any form, but an re$^{i\theta}$ entry causes an error in Degree mode. For polar entries the form (r∠θ) must be used.

**ref(***matrix*[,*tol*]**)**
Return the row echelon form of *matrix*. Treat any element as zero if it is less than *tol*, and the matrix contains floating-point elements and no symbolic elements. The default *tol* is 5E-14 * max(dim(*matrix*)) * rowNorm(*matrix*). Floating-point arithmetic is used in Approx mode.

**remain()**
(*expr1*,*expr2*), (*list1*,*list2*), (*matrix1*,*matrix2*)
Return the remainder of the first argument with respect to the second argument as
remain(x,0) = x
remain(x,y) = x-y*iPart(x/y)
Note that remain(-x,y) = -remain(x,y).

**Rename** *oldVarName*,*newVarName*
Rename variable *oldVarName* as *newVarName*.

**Request** *promptString*,*var*
If used inside a Dialog...EndDlog structure, create a user input box in the dialog box, otherwise create a dialog box with the input box. *promptString* must be less than 21 characters. If *var* contains a string, it is displayed as the default.

**Return** [*expression*]
Exit a program or function if no argument used, otherwise return *expression* from a function.

**right()**
(*list*[,*num*])
Return the rightmost *num* elements of *list*. Return *list* if *num* not used.

(*string*,[*num*])
Return the rightmost *num* characters of *string.* Return *string* if *num* not used.

(*comparison*)
Return the right side of equation or inequality.

**rotate()**
(*integer*,[*n*])
Rotate signed 32-bit *integer n* bits. Rotate to the left if *n* is positive, to the right if *n* is negative. Default *n* is -1.

(*list*,[*n*])
Return *list* with elements rotated by *n* elements. Rotate to the left for positive *n*, to the right for negative *n*. Default *n* is -1.

(*string*,[*n*])
Return *string* with characters rotated by *n* characters. Rotate to the left for for positive *n*, to the left for negative *n*. Default *n* is -1.

**round()**
(*expr*[,*d*]), (*list*[,*d*]), (*matrix*[,*d*])
Return argument elements rounded to *d* digits after the decimal point. *d* must be an integer in the range 0-12. Default *d* is 12.

**rowAdd(***matrix*,*r1*,*r2***)**
Return *matrix* with row *r1* replaced by the sum of rows *r1* and *r2*.

**rowDim(***matrix***)**
Return number of rows of *matrix*.

**rowNorm(**_matrix_**)**
Return the maximum of the sums of the row elements of _matrix_. All elements must simplify to numbers.

**rowSwap(**_matrix_,_r1_,_r2_**)**
Return _matrix_ with rows _r1_ and _r2_ swapped.

**RplcPic** _picVar_[,_row_][,_column_]
Place picture _picVar_ in the Graph screen with its upper left corner at pixel coordinates _row, column_. The area of the Graph screen affected by _picVar_ is cleared. _row_, _column_ default to 0,0.

**rref(**_matrix_[,_tol_]**)**
Return the reduced row echelon format of matrix. Treat any element as zero if it is less than _tol_, and the matrix contains floating-point elements and no symbolic elements. The default _tol_ is 5E-14 * max(dim(_matrix_)) * rowNorm(_matrix_). Floating-point arithmetic is used in Approx mode.

**Send** [_list_]
Send _list_ to the link port; used with CBL and CBR.

**SendCalc** _var_
Send _var_ to link port to be received by another calculator. Receiving calculator must be on Home screen or execute GetCalc from a program. Use SendChat, instead, to send from a TI-89/92+ to a TI-92.

**SendChat** _var_
Alternative to SendCalc which works with either a TI-92 or TI-92+. Will only send variables which are compatible with the TI-92. Will not send archived variables or graph data base, etc.

**seq(**_expr_,_var_,_low_,_high_[,_step_]**)**
Return list whose elements are _expr_ evaluated at _var,_ for each _var_ from _low_ to _high_ incremented by _step_. The default for _step_ is 1.

**setFold(**_folderName_**)**
Return the name of the current folder as a string, and set the current folder to _folderName_. The folder _folderName_ must exist.

**setGraph(**_modeNameString,settingString_**)**
Set the Graph mode _modeName_ to _setting_ and return the previous setting as a string. _modeNameString_ and _settingString_ may be descriptive strings, or numeric code strings. Numeric codes are preferred because the descriptive strings depend on the language localization setting. This table shows both the descriptive string (in English) followed by the equivalent numeric code.

| _modeNameString_ | _settingString_ |
|---|---|
| "Coordinates"<br>"1" | "Rect", "1"<br>"Polar", "2"<br>"Off", "3" |
| "Graph Order"<br>"2" | "Seq", "1"<br>"Simul" "2" [a] |
| "Grid"<br>"3" | "Off" "1"<br>"On" "2" [b] |
| "Axes"<br>"4" | Not 3D mode:<br>"Off" "1"<br>"On" "2"<br>3D mode:<br>"Off" "1"<br>"Axes" "2"<br>"Box" "3" |
| "Leading cursor"<br>"5" | "Off" "1"<br>"On" "2" |
| "Labels"<br>"6" | "Off" "1"<br>"On" "2" |
| "Seq Axes"<br>"7" | "Time" "1"<br>"Web" "2"<br>"U1-vsU2" "3" [d] |
| "Solution method"<br>"8" | "RK" "1"<br>"Euler" "2" [e] |
| "Fields"<br>"9" | "SlpFld" "1"<br>"DirFld" "2"<br>"FldOff" "3" [e] |
| "DE Axes"<br>"10" | "Time" "1"<br>"t-vs-y'" "2"<br>"y-vs-t" "3"<br>"y1-vs-y2" "4"<br>"y1-vs-y2'" "5"<br>"y1'-vs'y2'" "6" [e] |
| "Style"<br>"11" | "Wire Frame" "1"<br>"Hidden Surface" "2"<br>"Contour Levels" "3"<br>"Wire and Contour" "4"<br>"Implicit Plot" "5" [c] |

[a] Not available in Sequence, 3D or Diff Equations graph mode
[b] Not available in 3D graph mode
[c] Applies only to 3D graph mode
[d] Applies only to Sequence graph mode
[e] Applies only to Diff Equations graph mode

**setMode()**
(*modeString*,*settingString*), (*list*)
Set mode *modeString* to *settingString*, and return the current setting. *list* contains pairs of *modeString* and *settingString* pairs. *modeString* and *settingString* may be descriptive strings, or numeric code strings. Numeric codes are preferred because the descriptive strings depend on the language localization setting. Use the *list* argument to set several modes at once. This table shows both the descriptive string (in English) followed by the equivalent numeric code.

| *modeString* | *settingString* |
|---|---|
| "ALL"<br>"0" | Only applies to getMode(), not setMode() |
| "Graph"<br>"1" | "Function" "1"<br>"Parametric" "2"<br>"Polar" "3"<br>"Sequence" "4"<br>"3D" "5"<br>"Diff Equations" "6" |
| "Display digits"<br>"2" | "Fix 0", ..., "Fix 12"<br>"Fix n" is "n+1"<br>"Float" "14"<br>"Float 1", ..., "Float 12"<br>"Float n" is "n+14" |
| "Angle"<br>"3" | "Radian" "1"<br>"Degree" "2" |
| "Exponential Format"<br>"4" | "Normal" "1"<br>"Scientific" "2"<br>"Engineering" "3" |
| "Complex Format"<br>"5" | "Real" "1"<br>"Rectangular" "2"<br>"Polar" "3" |
| "Vector Format"<br>"6" | "Rectangular" "1"<br>"Cylindrical" "2"<br>"Spherical" "3" |
| "Pretty Print"<br>"7" | "Off" "1"<br>"On" "2" |
| "Split Screen"<br>"8" | "Full" "1"<br>"Top-Bottom" "2"<br>"Left-Right" "3" |
| "Split 1 App"<br>"9" | (no number codes)<br>"Home"<br>"Y= Editor"<br>"Window Editor"<br>"Graph"<br>"Table"<br>"Data/Matrix Editor"<br>"Program Editor"<br>"Text Editor"<br>"Numeric Solver"<br>"*Flash App*" |
| "Split 2 App"<br>"10" | Same as "Split 1 App" |
| "Number of Graphs"<br>"11" | "1" "1"<br>"2" "2" |
| "Graph2"<br>"12" | Same as "Graph" |
| "Split Screen Ratio"<br>"13" | "1:1" "1"<br>"1:2" "2"<br>"2:1" "3" (TI-92+ only) |
| "Exact/Approx"<br>"14" | "Auto" "1"<br>"Exact" "2"<br>"Approximate" "3" |
| "Base"<br>"15" | "Dec" "1"<br>"Hex" "2"<br>"Bin" "3" |
| "Language"<br>(no number code) | "English", "*Alternate Language"* |

**setTable(***modeNameString*,*settingString***)**
Set table parameter *modeNameString* to *settingString*, and return the previous setting. *modeNameString* and *settingString* may be descriptive strings, or numeric code strings. Numeric codes are preferred because the descriptive strings depend on the language localization setting. This table shows both the descriptive string (in English) followed by the equivalent numeric code.

| *modeNameString* | *settingString* |
|---|---|
| "Graph<->Table"<br>"1" | "Off" "1"<br>"On" "2" |
| "Independent"<br>"2" | "Auto" "1"<br>"Ask" "2" |

**setUnits(***list***)**
Set default units to values specified by *list* and return the previous defaults. Specify the built-in SI (metric) units with {"SI"}, or the English/US units with {"ENG/US"}.

Specify a custom set of default units with {"CUSTOM","*cat1*","*unit1*"[,"*cat2*","*unit2*",...]} where "*cat*" specifcies a unit category and "*unit*" specifies the default unit. Any unspecified category uses the previous custom unit.

Specify the previous custom default units with {"CUSTOM"}

**Shade**
*expr1*,*expr2*[,*xlow*][,*xhigh*][,*pattern*][,*patRes*]
Display Graph screen, graphs *expr1* and *expr2*, and shade areas where *expr1* < *expr2. expr1*

and *expr2* use *x* as the independent variable. *xlow* and *xhigh* specify left and right shading boundaries, *xlow* and *xhigh* are bounded by, and default to *xmin* and *xmax*.

*pattern* sets the shading pattern:
1 = vertical (default)
2 = horizontal
3 = negative-slope 45°
4 = positive-slope 45°

*patRes* specifies the shading pixel spacing resolution:
1 = solid
2 = 1 pixel spacing (default)
3 = 2 pixel spacing
...
10 = 9 pixel spacing

**shift()**
(*integer*[,*n*])
Shift bits in binary *integer*, *n* times. Shift to left if *n* is positive; to right if *n* is negative. Default *n* is -1. In a right shift, the rightmost bit is dropped, and 0 or 1 is inserted at the left to match the leftmost bit. In a left shift, the leftmost bit is dropped and 0 is inserted as the rightmost bit.

(*list*[,*n*])
Return *list* shifted left or right by *n* elements. Shift to left if *n* is positive, to right if *n* is negative. Default *n* is -1. Elements introduced by the shift are set to the symbol *undef*.

(*string*[,*n*])
Return *string* shifted left or right by *n* characters. Shift to left if *n* is positive, to right if *n* is negative. Default *n* is -1. Elements introduced by the shift are set to a space.

**ShowStat**
Show dialog box with the last computed statistics results, if the results are still valid. Results are cleared (not valid) if the data to compute them has changed.

**sign()**
(*expr*), (*list*), (*matrix*)
For real and complex arguments, return *expr*/abs(*expr*) when *expr* is not equal to zero. Return 1 if *expr* is positive, or -1 if *expr* is negative. sign(0) returns ±1 in the REAL complex mode, otherwise it returns itself.

**simult(***coefMatrix*,*constVector*[,*tol*]**)**
Return column vector with solutions to the system of linear equations where *coefMatrix* is a square matrix of the equation coefficients and *constVector* is a column vector with the same number of rows as *coefMatrix*. If the matrix has floating point elements and no symbolic variables, then any element is treated as zero if its value is less than *tol*. The default for *tol* is 5E-14 * max(dim(*coefMatrix*)) * rowNorm(*coefMatrix*). Computations are done with floating-point arithmetic in Approx mode.

(*coefMatrix*,*constMatrix*[,*tol*])
Solve multiple systems of linear equations, where each system has the same equation coefficients in *coefMatrix*. Each column of *constMatrix* contains the constants for a different system of equations. Each column of the returned matrix is the solution for the corresponding *constMatrix* column.

**sin()**
(*expr*), (*list*)
Return the sine of the argument. The argument is interpreted in degrees or radians according to the current mode setting. Use ° or $^r$ to override the mode setting.

(*matrix*)
Return the matrix sine of square *matrix*, which is not the sine of each element. Refer to cos() for details. *matrix* must be diagonalizable and the result always constains floating-point numbers.

**sin$^{-1}$()**
(*expr*), (*list*)
Return the angle whose sine is the argument. The result is returned as degrees or radians depending on the current Angle mode setting.

(*matrix*)
Return the matrix inverse sine of square *matrix*, which is not the same as the inverse sine of each element. Refer to cos() for details. *matrix* must be diagonalizable and the result always constains floating-point numbers.

**sinh()**
(*expr*), (*list*)
Return the hyperbolic sine of the argument. The argument is interpreted in degrees or radians according to the current mode setting. Use ° or $^r$ to override the mode setting.

(*matrix*)
Return the matrix hyperbolic sine of square *matrix*, which is not the hyperbolic sine of each element. Refer to cos() for details. *matrix* must be diagonalizable and the result always constains floating-point numbers.

**sinh$^{-1}$()**
(*expr*), (*list*)
Return the angle whose hyperbolic sine is the argument. The result is returned as degrees or radians depending on the current Angle mode setting.

(*matrix*)
Return the matrix inverse hyperbolic sine of square *matrix*, which is not the same as the inverse hyperbolic sine of each element. Refer to cos() for details. *matrix* must be diagonalizable and the result always constains floating-point numbers.

**SinReg**
*xlist*,*ylist*[,[*iterations*][,*period*][,*catList*,*incList*]]
Calculate the sinusoidal regression and update the system statistics variables. *xlist* and *ylist* are the x- and y-data points. *iterations* is the the maximum number of solution interations; the range is 1 to 16 and the default is 8. Larger values may result in better accuracy but longer execution time. *period* specifies the estimated period. If not used, the elements of *xlist* should be in sequential order and equally spaced. *xlist*, *ylist* and *catList* must be variable names or c1 - c99. *incList* need not be a variable name and cannot be c1 - c99. All lists must have equal dimensions except *incList*.

**solve()**
(*eqn*,*var*), (*inequality*,*var*)
Return candidate real solutions (as Boolean expressions) for *var* of equation *eqn* or *inequality*. Attempts to return all solutions, but for some arguments there are infinite solutions. In the Auto mode setting, concise exact solutions are attempted, supplemented by approximate solutions. Solutions may exist only in the limit from one or both sides due to default cancellation of the greatest common divisor from ratio numerators and denominators. *false* is returned if no real solutions can be found. *true* is returned if solve() determines that any finite real value is a solution. Solutions may include unique arbitrary integers of the form @n*j*, where *j* is an integer from 1 to 255. Use the "|" operator to constrain the solution interval or other variables.

In Real mode, fractional powers with odd denominators denote only the real branch. Otherwise, multiple branched expressions (fractional powers, logarithms, inverse trigonometric functions) denote the principle branch. solve() produces solutions only for that one real or principle branch.

Explicit solutions to inequalities are unlikely unless the inequality is linear and only includes *var*. In the Exact mode setting, portions which cannot be solved are returned as implicit equations or inequalities.

(*eqn1* and *eqn2* [and ...],{*var1*,*var2*[,...]})
Return candidate real solutions (as Boolean expressions) to the simultaneous equations. *var* arguments may be variable names, or variable names with a solution guess in the form *var* = *guess*. If all equations are polynomials and you supply no guesses, solve() uses the lexical Gröbner/Buchberger elimination to attempt to find all solutions. Simultaneous polynomial equations can have extra variables with no values.  Solution variables of no interest may be omitted. Solutions may include arbitrary constants of the form @*k*, where *k* is an integer from 1 to 255. Computation time or memory exhaustion may depend on the order of the *var*s in the equations or variables list.

solve() attempts to find all real solutions with Gaussian elimination if you include no guesses, any equation is in non-polynomial in any variable, but all equations are linear in the solution variables.

solve() attempts to find one real solution (with an interative approximate method) if the system is neither polynomial in all its variables nor linear in its solution variables. The number of solution variables must equal the number of equations and all other variables must simplify to numbers. Each solution variable starts at its guess value, or 0.0 if a guess is not used. Guesses may need to be close to the solution for convergence.

**SortA**
*listName1*[,*listName2*][,*listName3*],...

*vectorName1*[,*vectorName2*][,*vectorName3*],...
Sort the elements of the first argument in ascending order. Sort elements of optional arguments so that the element positions match the new positions of the elements in the first argument. All arguments must be names of lists or vectors, and have equal dimensions.

**SortD**
*listName1*[,*listName2*][,*listName3*],...
*vectorName1*[,*vectorName2*][,*vectorName3*],...
Same as SortA, but sort the elements in descending order.

**▸Sphere**
*vector* ▸Sphere
Display 3-element row or column *vector* in spherical form [ρ∠θ∠ϕ]. θ is the angle from the x-axis to the projection of the vector in the xy plane, and ϕ is the angle from the z-axis to the vector.

**stdDev(***list*[,*freqList*]**)**
Return the sample (not population) standard deviation of the elements of *list*. *freqList* elements specify the frequency of the corresponding elements of *list*. Both lists must have at least two elements, and the same number of elements.

(*matrix*[,*freqMatrix*])
Return row vector of the sample standard deviations of the column elements of *matrix*. *freqMatrix* elements specify the frequency of corresponding *matrix* elements. *matrix* must have at least two rows.

**StoGDB** *GDBvar*
Create a Graph database variable (GDB) with the current settings for:
Graphing mode; Y= functions; Window variables; Graph format settings (1- or 2-Graph setting; split-screen and ratio settings if 2-Graph mode); Angle mode; Real/Complex mode; Initial conditions (Sequence or Diff Equations mode); Table flags; tblStart, Δtbl, tblInput
Use RclGDB *GDBvar* to restore the graph environment.

**Stop**
Stop program execution.

**StoPic** *picVar*[,*pxlRow*,*pxlCol*][,*width*,*height*]
Display the Graph screen and copy a rectangular area of the display to *picVar*. *pxlRow* and *pxlCol* specify the upper left corner of the copied area, the default is 0,0. *width* and *height* specify the pixel dimensions of the area and default to the width and height of the current graph screen.

**Store**
See →

**String*(****expr***)***
Simplify *expr* and return the result as a string.

**Style** *eqNum*,*styleString*
Set graph function *eqNum* to use the graphing property *styleString*. *eqNum* must be 1 - 99 and the function must exist. *styleString* must be one of: "Line", "Dot", "Square", "Thick", "Animate", "Path", "Above" or "Below". Only some styles are valid for particular graph modes:

Function: all styles
Parametric/Polar: line, dot, square, thick, animate, path
Sequence: line, dot, square, thick
3D: none
Diff Equations: line, dot, square, thick, animate, path.

**subMat()**
(*matrix*[,*startRow*][,*startCol*][,*endRow*][,*endCol*])
Return specified submatix of *matrix*. Defaults are *startRow* = 1, *startCol* = 1, *endRow* = last row, *endCol* = last column

**sum(***list*[,*start*[,*end*]]**)**
Return sum of *list* elements from *start* to *end*.

(*matrix*[,*start*[,*end*]])
Return row vector whose elements are the sums of the columns of *matrix*. *start* and *end* specify start and end rows.

**switch(**[*n*]**)**
Return the number of the active window, and switch the active window based on *n*. Window 1 is top or left, window 2 is right or bottom. With no argument, switch the current window and return the previous active window number. *n* is ignored if a split screen is not displayed. Otherwise, for
*n* = 0: return the current window

*n* = 1: activate window 1, return previous active window number.
*n* = 2: activate window 2, return previous active window number.

**$^T$ (transpose)**
*matrix*$^T$
Return complex conjugate transpose of *matrix*.

**Table** *expr1*[,*expr2*][,*var*]
Build a table of the argument expression(s) or functions. The current Y= Editor functions are temporarily ignored. Expressions entered with Table or Graph are assigned increasing function numbers beginning with 1. To clear these functions execute ClrGraph or display the Y= Editor. If *var* is omitted, the current graph mode independent variable is used. Table cannot be used with 3D, sequence or differential equations graphing. Valid variations are:
Function graph: Table *expr*,*x*
Parametric graph: Table *xExpr*,*yExpr*,*t*
Polar graph: Table *expr*, θ
See also BldData.

**tan()**
(*expr*), (*list*)
Return the tangent of the argument. The argument is interpreted in degrees or radians according to the current mode setting. Use ° or $^r$ to override the mode setting.

(*matrix*)
Return the matrix tangent of square *matrix*, which is not the tangent of each element. Refer to cos() for details. *matrix* must be diagonalizable and the result always constains floating-point numbers.

**tan$^{-1}$()**
(*expr*), (*list*)
Return the angle whose tangent is the argument. The result is returned as degrees or radians depending on the current Angle mode setting.

(*matrix*)
Return the matrix inverse tangent of square *matrix*, which is not the same as the inverse tangent of each element. Refer to cos() for details. *matrix* must be diagonalizable and the result always constains floating-point numbers.

**tanh()**
(*expr*), (*list*)
Return the hyperbolic tangent of the argument. The argument is interpreted in degrees or radians according to the current mode setting. Use ° or $^r$ to override the mode setting.

(*matrix*)
Return the matrix hyperbolic tangent of square *matrix*, which is not the hyperbolic tangent of each element. Refer to cos() for details. *matrix* must be diagonalizable and the result always constains floating-point numbers

**tanh$^{-1}$()**
(*expr*), (*list*)
Return the angle whose hyperbolic tangent is the argument. The result is returned as degrees or radians depending on the current Angle mode setting.

(*matrix*)
Return the matrix inverse hyperbolic tangent of square *matrix*, which is not the same as the inverse hyperbolic tangent of each element. Refer to cos() for details. *matrix* must be diagonalizable and the result always constains floating-point numbers.

**taylor(***expr*,*var*,*order*[,*point*]**)**
Return the Taylor polynomial for *expr* in *var*. *point* is the expansion point and defaults to zero. The polynomial includes non-zero terms of integer degree from 0 through *order* in (*var* - *point*). taylor() returns itself if there is not truncated power series of this *order*, or negative or fractional exponents are required. Use substitution or temporary multiplication by a power of (*var* - *point*) for a more general series.

**tCollect(***expr***)**
Return expression in which products and integer powers of sines and cosines are converted to a linear combination of sines and cosines of multiple arguments, angle sums and angle differences. Trigonometric polynomials are converted to linear combinations of harmonics. tCollect() tends to reverse tExpand(). Applying tExpand() to a tCollect() result, or vice versa, in two steps may simplify an expression.

**tExpand(***expr***)**
Return expression in which sines and cosines of integer-multiple angles, angle sums and angle

differences are expanded. Best used in Radian mode, because degree-mode scaling interferes with expansion. tExpand() tends to reverse tCollect(). Applying tCollect() to a tExpand() result, or vice versa, in two steps may simplify an expression.

**Text** *string*
Display *string* in a dialog box. If used outside of a Dialog...EndDlog block, a dialog box is created, otherwise *string* is shown in the defined dialog box.

**Then**
See If.

**Title** *string*[,*label*]
Create the title of a menu or dialog box when used in a ToolBar or Custom structure, or a Dialog ... EndDlog block. *label* is only valid in the ToolBar structure. When used, it allows the menu choice to branch to *label*.

**tmpCnv(***expr_°unit1,_°unit2***)**
Return temperature value *expr* converted from *unit1* to *unit2*. Valid units are _°C, _°F, _°K and _°R. To convert a temperature range instead of a value, use ΔtmpCnv().

**ΔtmpCnv(***expr_°unit1,_°unit2***)**
Return temperature range *expr* converted from *unit1* to *unit2*. Valid units are _°C, _°F, _°K and _°R. To convert a temperature value instead of a range, use tmpCnv().

**Toolbar**
ToolBar : *block* : EndTBar
Create a toolbar menu. *block* statements can be either Title or Item. Item statements must have labels. Title statements must have labels if there are no Item statements.

**Trace**
Draw a SmartGraph and place the trace cursor on the first Y= function, at the previous cursor position, or at the reset position if regraphing was necessary. Allows operation of the cursor keys and most coordinate editing keys. Press [ENTER] to resume operation.

**Try**
Try : *block1* : Else : *block2* : EndTry
Execute *block1* until an error occurs, then transfer execution to *block2*. *errornum* contains the error number. See also ClrErr and PassErr.

**TwoVar** *xList,yList*[[,*freqList*][,*catList*][,*incList*]]
Calculate two-variable statistics and update system statistics variables. All lists must have equal dimension except *incList*. The first four arguments must be variable names of c1 - c99. *incList* need not be a variable name and cannot be c1 - c99.

**Unarchiv** *var1*[,*var2*][,*var3*]...
Move argument variables from user data archive memory (flash) to RAM. Archived variables can be accessed, but not deleted, renamed or stored to, since the variable is locked. Use Archive to archive variables.

**unitV(***vector***)**
Return row- or column-unit vector, depending on the form of *vector*, which must be a single-row or single-column matrix.

**Unlock** *var1*[,*var2*][,*var3*]...
Unlock the argument variables. Use Lock to lock variables.

**variance(***list*[,*freqList*]**)**
Return the population variance of the elements of *list*. *freqList* elements specify the frequency of the corresponding elements of *list*. Both lists must have at least two elements, and the same number of elements.

(*matrix*[,*freqMatrix*])
Return row vector of the population variance of the column elements of *matrix*. *freqMatrix* elements specify the frequency of corresponding *matrix* elements. *matrix* must have at least two rows.

**when()**
(*condition*,*trueRes*[,*falseRes*][,*unknownRes*])
Return *trueRes*, *falseRes*, or *unknownRes*, when *condition* is true, false or unknown. Returns the arguments when there too few arguments to determine the result. Omit both *falseRes* and *unknownRes* to define an expression only when *condition* is true. *undef* can be used as a result.

**While**
While *condition* : *block* : EndWhile

Execute *block* as long as *condition* is true.

**"With"** See |

**xor**
*exp1* xor *exp2*
*list1* xor *list2*
*matrix 1* xor *matrix 2*
Return *true*, *false* or a simplifed form.

*integer1* and *integer2*
Bit-by-bit 32-bit integer logical exclusive-or. Arguments larger than 32-bit signed values are reduced with symmetric modulo operation. Mode must be Auto or Exact.

**XorPic** *picVar*[,*row*][,*column*]
Display Graph screen and logically exclusive-or Graph screen pixels with those of *picVar* picture variable. Only pixels exclusive to the screen or *picVar* are turned on. *row* and *column* specify the pixel coordinates for the upper left corner of *picVar*; defaults are 0,0.

**zeros()**
(*expr*,*var*)
Return list of candidate values for *var* which make *expr* = 0. Equivalent to exp▸list(solve(*expr*=0,*var*))
zeros() cannot express implicit solutions, solutions which require inequalities or solutions which do not involve *var*.

({*expr1*,*expr2*[,*expr3*...]},{*var1*,*var2*[,*var3*...]})
Return matrix of candidate solutions of simultaneous algebraic expressions *expr* in *var*. *var* may be a variable or a solution guess in the form *var* = *guess*. Each matrix row represents an alternate solution, with the variables ordered as in the {*var*} list.

If all equations are polynomials and you supply no guesses, zeros() uses the lexical Gröbner/Buchberger elimination to attempt to find all solutions. Simultaneous polynomial equations can have extra variables with no values. Solution variables of no interest may be omitted. Solutions may include arbitrary constants of the form @*k*, where *k* is an integer from 1 to 255. Computation time or memory exhaustion may depend on the order of the *var*s in the equations or variables list.

zeros() attempts to find all real solutions with Gaussian elimination if you include no guesses, any equation is in non-polynomial in any variable, but all equations are linear in the solution variables.

zeros() attempts to find one real solution (with an interative approximate method) if the system is neither polynomial in all its variables nor linear in its solution variables. The number of solution variables must equal the number of equations and all other variables must simplify to numbers. Each solution variable starts at its guess value, or 0.0 if a guess is not used. Guesses may need to be close to the solution for convergence.

**ZoomBox**
Display the Graph screen, pauses so that a box can be drawn to define the new view window, then updates the window.

**ZoomData**
Adjust the window settings based on the current data plots and function graphs so that all data points are sampled, then displays the Graph Screen. Does not adjust *ymin* and *ymax* for histograms.

**ZoomDec**
Set Δx and Δy = 0.1 and displays the Graph screen with the origin centered.

**ZoomFit**
Display the Graph screen and set the dependent variable Window dimensions so that all the picture is displayed for the current independent variable settings.

**ZoomIn**
Display the Graph screen, pauses so that a center point can be set to zoom in, then updates the viewing window. The zoom magnitude depends on the Zoom factors *xFact* and *yFact* in 2D graph modes, and on *xFact*, *yFact* and *zFact* in 3D mode.

**ZoomInt**
Display the Graph screen, pauses so that a zoom center point can be set, adjusts the Window settings such that each pixel is an integer in all directions, then updates the viewing window.

**ZoomOut**
Display the Graph screen, pauses so that a zoom center point can be set, then updates the viewing window. The zoom magnitude depends on the Zoom factors *xFact* and *yFact* in 2D graph modes, and on *xFact*, *yFact* and *zFact* in 3D mode.

**ZoomPrev**
Display the Graph screen and update the viewing window with the settings before the previous zoom.

**ZoomRcl**
Display the Graph screen and update the viewing window with the settings stored with ZoomSto.

**ZoomSqr**
Display the Graph screen and adjust the x or y window settings so that each pixel represents an equal width and height in the coordinate system, then update the viewing window. In 3D mode, lengthen the shortest two axes to equal the longest axis.

**ZoomStd**
Set the Window variables to the following standard values, then update the viewing window.
Function graphing:
  x:[-10,10,1], y:[-10,10,1], *xres* = 2
Parametric graphing:
  t:[0,2π,π/24], x:[-10,10,1], y:[-10,10,1]
Polar graphing:
  θ:[0,2π,π/24], x:[-10,10,1], y:[-10,10,1]
Sequence graphing:
  x:[-10,10,1], y:[-10,10,1]
  *nmin*=1, *nmax*=10, *plotStrt*=1, *plotStep*=1
3D graphing:
  x:[-10,10,14], y:[-10,10,14], z[-10,10]
  eyeθ°=20, eyeϕ°=70, eyeψ°=0, *ncontour*=5
Differential equations graphing:
  t:[0,10,.1,0], x:[-1,10,1], y:[-10,10,1]
  *ncurves*=0, *Estep*=1, *diftol*=.001, *fldres*=14,
  *dtime*=0

**ZoomSto**
Store the current Window settings in Zoom memory. Use ZoomRcl to restore the settings.

**ZoomTrig**
Display the Graph screen, set x = π/24, xscl = π/2, center the origin, set the y settings to [-4,4,.5] and update the viewing window.

**+ (add)**
*expr1* + *expr2*
Return the sum of *expr1* and *expr2*

*list1* + *list2*
*matrix1* + *matrix2*
Return list or matrix whose elements are the sum of the corresponding elements. Argument dimensions must be equal.

*expr* + *list*
*list* + *expr*
Return list whos elements are the sum of the list elements and *expr*.

*expr* + *matrix*
*matrix* + *expr*
Return matrix with *expr* added to each diagonal element of square *matrix*. Use .+ (dot plus) to add an expression to each element.

**- (subtract)**
*expr1* - *expr2*
Return *expr1* - *expr2*

*list1* - *list2*
*matrix1* - *matrix2*
Return list or matrix whose elements are the elements of *list2* (or *matrix2*) subtracted from the corresponding elements of *list1* (or *matrix1*). Argument dimensions must be equal.

*expr* - *list*
*list* - *expr*
Return list where each element is each *list* element subtracted from *expr, or expr* subtracted from each list element.

*expr* - *matrix*
Return matrix: (*expr* * *identityMatrix*) - *matrix*
*matrix* must be square.

*matrix* - *expr*
Return matrix: *matrix* - (*expr* * *identityMatrix*)
*matrix* must be square.

(Note: use .- (dot minus) to subtract an expression from each element.)

*** (multiply)**
*expr1 * expr2*
Return the product of *expr1* and *expr2*

*list1 * list2*
Return list whose elements are the products of the corresponding elements of *list1* and *list2*. List dimensions must be equal.

*matrix1 * matrix2*
Return the matrix product of *matrix1* and *matrix2*. The number of rows of *matrix1* must equal the number of columns of *matrix2*.

*expr * list*
*list * expr*
Return list where each element is each *list* element multiplied by *expr*.

*expr * matrix*
*matrix * expr*
Return matrix whose elements are the product of *expr* and each element of *matrix*. This is the same as .* (dot multiply).

**/ (divide)**
*expr1* / *expr2*
Return the quotient of *expr1* divided by *expr2*

*list1* / *list2*
Return list whose elements are the quotients of the corresponding elements of *list1* and *list2*. List dimensions must be equal.

*expr* / *list*
*list* / *expr*
Return list where each element is the quotient of *expr* divided by each *list* element, or each *list* element divided by *expr*.

*matrix* / *expr*
Return matrix whose elements are the quotient of each element of *matrix* divided by *expr*. Use ./ (dot divide) to divide an expression by each matrix element.

**- (negate)**
-*expr*
-*list*
-*matrix*
Return the negation of the argument. If the argument is a binary or hexadecimal integer, return the two's-complement.

**% (percent)**
*expr*%
*list*%
*matrix*%
Return (argument/100)

**= (equal)**
*expr1 = expr2*
*list1 = list2*
*matrix1 = matrix2*
Return True if first argument can be determined to be equal to second argument. Return False if first argument cannot be determined to be equal to second argument. Otherwise, return a simplified form of the equations. For list and matrix arguments, return element-by-element comparisons.

**≠ (not equal)**
*expr1* ≠ *expr2*
*list1* ≠ *list2*
*matrix1* ≠ *matrix2*
Return True if first argument can be determined to be not equal to second argument. Return False if first argument can be determined to be equal to second argument. Otherwise, return a simplified form of the inequality. For list and matrix arguments, return element-by-element comparisons.

**< (less than)**
*expr1 < expr2*
*list1 < list2*
*matrix1 < matrix2*
Return True if first argument can be determined to be less than second argument. Return False if first argument can be determined to be greater than or equal to the second argument. Otherwise, return a simplified form of the inequality. For list and matrix arguments, return element-by-element comparisons.

**≤ (less than or equal to)**
*expr1 ≤ expr2*
*list1 ≤ list2*
*matrix1 ≤ matrix2*
Return True if first argument can be determined to be less or equal to the second argument. Return False if first argument can be determined to be greater than the second argument. Otherwise, return a simplified form of the comparison. For list and matrix arguments, return element-by-element comparisons.

**> (greater than)**
*expr1 > expr2*
*list1 > list2*
*matrix1 > matrix2*
Return True if first argument can be determined to be greater than the second argument. Return False if first argument can be determined to be less than or equal to the second argument. Otherwise, return a simplified form of the comparison. For list and matrix arguments, return element-by-element comparisons.

**≥ (greater than or equal to)**
*expr1 ≥ expr2*
*list1 ≥ list2*
*matrix1 ≥ matrix2*
Return True if first argument can be determined to be greater than or equal to the second argument. Return False if first argument can be determined to be less than the second argument. Otherwise, return a simplified form of the comparison. For list and matrix arguments, return element-by-element comparisons.

**.+ (dot add)**
*matrix1* .+ *matrix2*
Return matrix whose elements are the sums of the corresponding elements of *matrix1* and *matrix2*.

*expr* .+ *matrix*
*matrix* .+ *expr*
Return matrix whose elements are the sums of *expr* and *matrix.*

**.- (dot subtract)**
*matrix1* .- *matrix2*
Return matrix whose elements are the corresponding elements of *matrix2* subtracted from *matrix1*.

*expr* .- *matrix*
Return matrix whose elements are the elements of *matrix* subtracted from *expr*.

*matrix* .- *expr*
Return matrix whose elements are *expr* subtracted from the elements of *matrix.*

**.* (dot multiply)**
*matrix1 .* matrix2*
Return matrix whose elements are the products of each corresponding elements of *matrix1* and *matrix2*.

*expr* .* *matrix*
*matrix* .* *expr*
Return matrix whose elements are the product of *expr* and the elements of *matrix*.

**./ (dot divide)**
*matrix1 .* matrix2*
Return matrix whose elements are the quotients of the corresponding elements of *matrix1* divided by *matrix2*.

*expr* .* *matrix*
*matrix* .* *expr*
Return matrix whose elements are the quotients of *expr* divided by the elements of *matrix*, or the elements of *matrix* divided by *expr*.

**.^ (dot power)**
*matrix1 .^ matrix2*
Return matrix whose elements are the elements of *matrix1* raised to the power of the corresponding *matrix2* elements.

*expr* .^ *matrix*
*matrix* .^ *expr*
Return matrix whose elements are *expr* raised to the power of each *matrix* element, or the elements of *matrix* raised to the *expr* power.

**! (factorial)**
*expr*!
*list*!
*matrix*!
Return the factorial of the argument. Only returns a numeric value for non-negative integer arguments.

**& (append)**
*string1* & *string2*
Return string which is *string2* appended to *string1*.

**∫() (integrate)**
(*expr*,*var*[,*lower*][,*upper*])
(*list*,*var*[,*lower*][,*upper*])
(*matrix*,*var*[,*lower*][,*upper*])
Return the integral of the first argument with respect to *var*, from *lower* to *upper* integration limits. Return anti-derivative if *lower* and *upper* are omitted; constant of integration is omitted, but *lower* will be added as a constant of integration if *upper* is not used. Anti-derivatives may differ by a numeric constant. Piece-wise

constants may be added so that the anti-derivative is valid over a larger interval.

∫() can be nested for multiple integrals, and the integration limits can depend on integration variables outside the limits.

When both *lower* and *upper* are present, ∫() attempts to subdivide the integral at discontinuities. Numerical integration is used in Auto mode when a symbolic anti-derivative cannot be determined. Numerical integration is tried first in Approx mode, but anti-derivatives are sought if numerical differentiation is inapplicable or fails.

∫() returns itself for pieces of *expr* it cannot determine.

**√() (square root)**
√(*expr*), √(*list*)
Return the square root of the argument.

**Π(***expr*,*var*,*low*,*high***) (product)**
Evaluate *expr* for each value of *var* from *low* to *high*; return the product of the results. Return 1 if *high* = *low* -1. If *high* < *low* -1, return 1 / Π(*expr*,*var*,*high*+1,*low*-1)

**Σ(***expr*,*var*,*low*,*high***) (sum)**
Evaluate *expr* for each value of *var* from *low* to *high* and return the sum of the results. Return zero if *high* = *low* -1. If *high* < *low* -1, return -Σ(*expr*,*var*,*high*+1,*low*-1)

**^ (power)**
*expr1^expr2*
Return *expr1* raised to the power of *expr2*.

*list1^list2*
Return list whose elements are the elements of *list1* raised to the powers of the corresponding elements of *list2*.

*expr^list*
Return list whose elements are *expr* raised to the powers of the elements of *list*.

*list^expr*
Return list whose elements are the elements of *list* raised to the *expr* power.

*matrix^integer*
Return *matrix* raised to the *integer* power. *matrix* must be square. If *integer* = -1, return the matrix inverse. If *integer* < -1, return the inverse matrix raised to the *-integer* power.

**# (indirection)**
# *varNameString*
Refer to the variable whose name is *varNameString*.

**$^{r}$ (radian)**
*expr*$^{r}$, *list*$^{r}$, *matrix*$^{r}$
In Degree mode, multiply the argument by 180/π. Return argument unchanged in Radian mode. Use $^{r}$ to force radians regardless of the current Angle mode.

**° (degree)**
*expr*°, *list*°, *matrix*°
In Radian mode, multiply the argument by π/180. Return argument unchanged in Degree mode. Use ° to force radians regardless of the current Angle mode.

**∠ (angle)**

| | |
|---|---|
| [*r*,∠θ_*angle*] | (polar input) |
| [*r*,∠θ_*angle*,*z*] | (cylindrical input) |
| [*r*,∠θ_*angle*,ϕ_*angle*] | (spherical input) |

Return argument coordinates as a vector depending on the current Vector format setting: rectangular, cylindrical or spherical

| | |
|---|---|
| (*magnitude*∠*angle*) | (polar input) |

Enter a complex value in (r∠θ) format. *angle* is interpreted according to the current Angle mode setting.

**°, ', " (degree, minute, second)**
*dd*°*mm*'*ss.ss*"
Return *dd* + (*mm*/60) + (*ss.ss*/3600), where *dd* is may be positive or negative, *mm* and *ss.ss* are non-negative numbers. Enter a number in base-60 format, which allows entering angle in degrees, minutes and seconds, regardless of the current Angle mode. Also allows entry of times as hours, minutes and seconds.

**' (prime)**
*var*'
*var*''
Enter prime symbol in a differential equation (DE). A single prime denotes a first-order DE, two prime symbols denote a second-order DE.

**_ (underscore)**
*expr_unit*
Designate the *unit*s for *expr*. All unit names begin with the underscore.
*var_*
Treat *var* as complex if it has no value. By default, variables without the underscore are treated as real. Variables with values are treated as the value type; real or complex. Complex numbers can be stored in variables without the underscore, but complex operations work better if underscores are used.

**▸ (convert)**
*expr_unit1*▸*_unit2*
Convert *expr* with units *unit1* to units *unit2*, excluding temperature conversions. The units must be in the same category. Use tmpCnv() and ΔtmpCnv() for temperature conversions.

**10^()**
(*expr*), (*list*)
Return 10 raised to the power of the argument.

(*matrix*)
Return 10 raised to the power of square *matrix*. This is not the same as 10 raised to the power of each element. Refer to cos() for calculation details. *matrix* must be diagonalizable, and the result always contains floating-point numbers.

**$x^{-1}$ (^-1)**
*expr*^-1
*list*^-1
Return the reciprocal of the argument.

*matrix*^-1
Return the inverse of square *matrix*, which must be non-singular.

**| ("with")**
*expr*|*Boolean1* [and *Boolean2*]...[and *BooleanN*]
Evaluate *expr* subject to the *Boolean* contraints. This allows substitutions, interval constraints and exclusions. If *Boolean* is an equality such as *var* = *value*, then each occurence of *var* in *expr* is substituted with *value*. A constraint is formed with two or more *Boolean*s joined with 'and', where each *Boolean* is an inequality. An exclusion is a *Boolean* of the form *var ≠ value*, which is primarily used to exclude exact solutions of cSolve(), cZeros(), solve(), etc.

**→ (store)**
*expr→var*
*list→var*
*matrix→var*
Create *var* if it does not exist and initialize it to *expr*, *list* or *matrix*. If *var* exists and is not locked or protected, replace its contents with *expr*, *list* or *matrix*.

*expr→functionName*([*parameter1*,...])
Create *functionName* if it does not exist and initialize it to *expr*. If *functionName* exists and is not locked or protected, replace its contents with *expr*. *parameter*s are optional function arguments.

*list→functionName*([*parameter1*,...])
*matrix→functionName*([*parameter1*,...])
Create *functionName* if it does not exist and initialize it to *list* or *matrix*. If *functionName* exists and is not locked or protected, replace its contents with *list* or *matrix*. *parameter*s are optional function arguments.

**© (comment)**
[*text*]
Process *text* as a comment line in a program or function. © may be at the beginning of a line or anywhere within the line. All *text* to the right of © to the end of the line is the comment.

**0b, 0h**
0b*binaryNumber*
0h*hexadecimalNumber*
Denote a binary or hexadecimal integer. 0b or 0h must be entered regardless of the Base mode setting, otherwise the number is treated as decimal (base 10).

## Reserved system variable names

Δtbl
Δx
Δy
Σx
$\Sigma x^2$
Σxy
Σy
$\Sigma y^2$
σx
σy
θc
θmax
θmin
θstep
c1-c99
corr
diftol
dtime
eqn
errornum
estep
exp
eyeθ
eyeϕ
eyeψ
fldpic
fldres
main
maxX
maxY
medStat
medx1
medx2
medx3
medy1
medy2
medy3
minX
minY
nc
ncontour
ncurves
nmax
nmin
nStat
ok
plotStep
plotStrt
q1
q3
$R^2$
r1()-r99()
rc
regCoef
regEq(x)
seed1
seed2
Sx
Sy
sysData
sysMath
tØ
tblInput
tblStart
tc
tmax
tmin
tplot
tstep
u1()-u99()
ui1-ui99
$\bar{x}$
xc
xfact
xgrid
xmax
xmin
xres
xscl
xt1()-xt99()
$\bar{y}$
y1'()-y99'()
y1()-y99()
yc
yfact
ygrid
yi1-yi99
ymax
ymin
yscl
yt1()-yt99()
zθmax
zθmin
zθstep
z1()-z99()
zc
zeyeθ
zeyeϕ
zeyeψ
zfact
zmax
zmin
znmax
znmin
zplstep
zplstrt
zscl
ztØde
ztmax
ztmaxde
ztmin
ztplotde
ztstep
ztstepde
zxgrid
zxmax
zxmin
zxres
zxscl
zygrid
zymax
zymin
zyscl
zzmax
zzmin
zzscl

## EOS (Equation Operating System) Hierarchy

Exponentiation (^) and element-by-element exponentiation (.^) are evaluated from right to left, for example, 2^3^4 = 2^(3^4).

Post operations and exponentiation are performed before negation, for example, -9^2 = -(9^2).

Other operations are performed according the these priorities:

1 Parentheses (), brackets [] and braces {}
2 Indirection (#)
3 Function calls
4 Post operators: degrees-minutes-seconds (°,',"), factorial (!), percentage (%), radian ($^r$), transpose ($^T$)
5 Exponentiation, power operator (^)
6 Negation (-)
7 String concatenation (&)
8 Multiplication (*), division (/)
9 Addition (+), subtraction (-)
10 Equality relations: =, ≠, <, ≤, >, ≥
11 Logical 'not'
12 Logical 'and'
13 Logical 'or', 'xor'
14 Constraint "with" operator (|)
15 Store (→)

# Appendix E: Tip List programs and functions

This appendix lists the functions and programs used in the tip list. The files in the *tlcode.zip* file have the same names as those shown, but the file type may be .89p, .9xp, .89f and so on, depending on the original calculator file type. For some programs there are specific versions for the TI-89 and the TI-92 Plus, and these can be identified by the file types .89* and .9x*. Unless otherwise stated in the tip description, the routines will run on either calculator.

| ***Name*** | ***Description*** | ***Tip*** |
|---|---|---|
| adjoint() | Matrix adjoint | [3.13] |
| aitkend2() | Accelerate series convergence with Aitken's method | [6.59] |
| apps() | Application launcher (DAB) | [7.18] |
| asympexp() | Asymptotic expansion | [6.61] |
| atan2() | Arc-tangent, four-quadrant | [6.48] |
| autoexap() | Toggle Exact, Auto, Approx modes | [9.14] |
| bilinint() | Bilinear interpolation | [6.28] |
| bilinui() | Bilinear interpolation, user interface for bilinint() | [6.28] |
| bitclr() | Clear a bit in a word | [3.23] |
| bitnot() | Invert a bit in a word | [3.23] |
| bitset() | Set a bit in a word | [3.23] |
| bittst() | Test a bit in a word | [3.23] |
| bn() | Bernoulli number | [6.27] |
| bnpoly() | Bernoulli polynomial; table of polynomials | [6.27] |
| bnpolys() | Bernoulli polynomial of order n; single polynomial | [6.27] |
| bpo() | Bernoulli polynomial; recall polynomial, see bnpoly() | [6.27] |
| c2p() | Convert expression in rectangular coordinates to polar | [6.15] |
| casel() | Convert string to lower-case | [8.6] |
| caseu() | Convert string to upper-case | [8.6] |
| charcnt() | Count character occurrences in string | [8.4] |
| ci() | Cosine integral function | [6.33] |
| cnd() | Normal distribution, cumulative, with error function | [6.35] |
| cndi() | Normal distribution inverse, accurate version | [6.35] |
| cndif() | Normal distribtution inverse, fast version | [6.35] |
| cndint() | Normal distribution, cumulative, by integral definition | [6.35] |
| coef() | See cubic() and quartic() | [6.10] |
| copyto_h() | Copy expression to home screen (Samuel Stearley) | [7.8] |
| corrhk() | Correlation coefficient for regression through fixed point | [6.6] |
| cubic() | Exact solution to cubic polynomial | [6.10] |
| datalist() | Write values to data variables | [3.8] |
| datamat() | Convert data variable to matrix | [3.29] |
| dbdemo() | Dialog box (dynamic) demonstration | [9.7] |
| dbdrd() | Dialog box (dynamic), drop-down menu | [9.7] |
| dbend() | Dialog box (dynamic), end | [9.7] |
| dbreq() | Dialog box (dynamic), Request | [9.7] |
| dbttl() | Dialog box (dynamic), title | [9.7] |
| dbtxt() | Dialog box (dynamic), text | [9.7] |
| delay() | Delay expression evaluation with constraints | [2.18] |
| deleqnv() | Delete variables in numeric solver equation | [6.54] |
| delvar1() | Delete locked, archived variables | [7.39] |
| det2a() | Fast symbolic matrix determinant | [3.27] |
| disprc() | Display text on program I/O screen | [9.4] |
| dms() | Convert angle in D.MS format to decimal degrees or radians | [6.17] |
| dmsi() | Convert angle in decimal degrees or radians to D.MS format | [6.17] |

| | | |
|---|---|---|
| ei() | Exponential integral | [6.60] |
| eqlib() | Equation library; single-menu | [9.21] |
| eqlibx() | Equation library; multiple menus | [9.21] |
| erf() | Error function | [6.34] |
| evalrt() | See cubic() and quartic() | [6.10] |
| exprvars() | Extract variable names from expression | [7.42] |
| f1demo() | Evaluate function with expression, list or matrix argument | [6.40] |
| factrl() | Factorial; non-integer and complex arguments | [6.4] |
| fib() | Fibonacci number | [6.24] |
| fill_lm() | Fill list or matrix with a constant | [3.28] |
| fipoly() | Fast floating-point polynomial solution | [6.12] |
| frctst() | Test fractions for equality | [2.2] |
| gamma() | Gamma function | [6.4] |
| graphClr() | Clear graph screen completely | [4.14] |
| grpfrc() | Group fraction digits | [9.10] |
| grpfrcan() | Group fraction digits of last ANS | [9.10] |
| iconex() | Toolbar icon example | [9.1] |
| im1a.9xm | Matrix used with intrpz9() | [6.28] |
| impdifn() | Implicit derivative | [6.53] |
| indexlst() | Return list indices for target element [TI Basic version] | [3.26] |
| indexmat() | Return matrix indices for target element | [3.26] |
| inlist() | Return list indices for target element (C version) | [3.26] |
| interp() | Linear interpolation | [6.22] |
| intrp9ui() | Nine-point 3-dimensional interpolation, user interface for intrpz9() | [6.28] |
| intrpz9() | Nine-point 3-dimensional interpolation | [6.28] |
| inv() | Matrix inverse, fast numeric | [6.63] |
| linreghk() | Linear regression through fixed point | [6.6] |
| linterp() | Linear interpolation with numeric solver | [6.22] |
| listrev() | Reverse list elements | [3.4] |
| listswap() | Swap list elements | [3.18] |
| lngamma() | Log-gamma function | [6.4] |
| loglist() | Make list with logarithmic element spacing | [3.22] |
| lstident() | Test list for identical elements | [3.19] |
| m3aind() | Return address of element of rank-3 array | [3.16] |
| m3rcl() | Recall element from 3-dimensional array | [3.16] |
| m3sto() | Store element to 3-dimensional array | [3.16] |
| m4aind() | Return address of element of rank-4 array | [3.16] |
| m4rcl() | Recall element from 4-dimensional array | [3.16] |
| m4sto() | Store element to 4-dimensional array | [3.16] |
| mantexp() | Extract mantissa and exponent from floating-point number | [6.58] |
| mat2cvec() | Convert matrix to single-column vector | [3.21] |
| mat2rvec() | Convert matrix to single-row vector | [3.21] |
| mid1() | Return string or list middle elements based on position limits | [8.7] |
| mminor() | Matrix minor | [3.13] |
| mnaind() | Return address of element of rank-n array | [3.16] |
| modecycl() | Cycle Mode setting to next option | [9.14] |
| model() | Calculator model | [7.14] |
| mrankni() | Return list index of element of n-dimensional array | [3.16] |
| mrowdel() | Delete matrix row | [3.3] |
| nder1() | Numerical derivative; Ridder's method | [6.26] |
| nder1p() | Numerical derivative; diagnostic routine for nder1() | [6.26] |
| nder2() | Numerical derivative; optimum accuracy from nDeriv() | [6.26] |
| nderui() | Numerical derivative; user interface for nder1() | [6.26] |
| nintx() | Numerical integration with intervals for better accuracy | [6.19] |
| p2c() | Convert expression in polar coordinates to rectangular | [6.15] |
| parcurve() | Convert equation to parameterized form | [6.39] |

| | | |
|---|---|---|
| pid() | Calculator product ID | [7.14] |
| plotdemo() | Plot data and functions simultaneously | [4.4] |
| plotintg() | Fast integral function plot | [4.10] |
| plotinui() | Fast integral function plot; user interface | [4.10] |
| plotxy() | Fast function plot | [4.9] |
| pnttrian() | Determine if a point is inside a triangle | [6.45] |
| polar() | Convert polar vector to rectangular | [6.25] |
| polyc() | Extract polynomial coefficients from expression | [6.61] |
| polyroot() | Polynomial roots with Laguerre's method | [6.61] |
| prooti() | Polynomial roots, coefficient inversion | [6.61] |
| proots() | Polynomial roots with eigenvalue method | [6.61] |
| proot_t1() | Polynomial root sign test | [6.61] |
| quadint() | Quadratic interpolation | [6.31] |
| quadrtic() | Quadratic equation; accurate solution with large coefficients | [6.32] |
| quartic() | Exact solution to quartic polynomial | [6.10] |
| r2coef() | Find coefficient of determination for all regression equations | [6.13] |
| r2coefdf() | r2coef() with degree-of-freedom correction | [6.13] |
| randlist() | List of random unique integers | [3.14] |
| rclexpr() | Recall expression without variable value substitution | [7.40] |
| rect() | Convert rectangular vector to polar | [6.25] |
| req2var() | Get two variables from one dialog box Request | [9.12] |
| req3var() | Get three variables from one dialog box Request | [9.12] |
| right1() | Return string or list right elements based on position limits | [8.7] |
| rms() | RMS statistic for list or matrix | [6.14] |
| rq2v() | Convert "a,b" to {a,b} | [9.12] |
| rq3v() | Convert "a,b,c" to {a,b,c} | [9.12] |
| ruler() | On-screen ruler and protractor | [1.16] |
| savage() | Savage floating-point benchmark, C version | [6.50] |
| savage2() | Savage floating-point benchmark, TI Basic version | [6.50] |
| scatterf() | Random scatter plot of function | [4.12] |
| sgn() | Sign function, compatible with other programming languages | [7.30] |
| si() | Sine integral function | [6.33] |
| sigdig() | Round number to significant digits | [6.5] |
| simulti() | Simultaneous equation solution with improved accuracy | [6.3] |
| solvebug() | Demonstrate nsolve() Questionable Accuracy message | [11.9] |
| solvemul() | Solve multiple equations in a program | [11.4] |
| spli4de() | 4th-order splice; derivatives | [6.56] |
| spli4in() | 4th-order splice; integrals | [6.56] |
| spli4inv() | 4th-order splice; inverse function | [6.56] |
| spli4ui() | 4th-order splice; user interface | [6.56] |
| spli4x() | 4th-order splice; inverse function at a point | [6.56] |
| splice4() | 4ht-order splice; find splice function coefficients | [6.56] |
| stddevp() | Standard deviation, population (not sample) | [6.42] |
| stddevpf() | Standard deviation, population, with frequencies | [6.42] |
| stoexact() | Convert floating-point number to exact fraction | [6.9] |
| str2var() | Convert 2-element string "x,y" to list {x,y} | [6.31] |
| strcompr() | Compare 2 strings; return character code differences | [5.8] |
| strsub() | String substitution | [8.2] |
| svd() | Singular value decomposition of matrix | [3.30] |
| TI Basic Extension Template.c | Demo template to add TI Basic extension to SDK flash app | [12.3 |
| tohome() | Copy expression to home screen (Timite Hassan) | [7.8] |
| tooltab() | Custom toolbar menu with key traps; demo | [7.46] |
| transpos() | Transpose rank-3 array | [3.16] |
| truth() | Truth plot | [4.3] |

| | | |
|---|---|---|
| truthd() | Truth plot with control arguments | [4.3] |
| units() | Add units to built-in Units menu | [10.6] |
| variancp() | Variance for population, not sample | [6.42] |
| varianpf() | Variance for population, not sample, with frequencies | [6.42] |
| version() | Calculator AMS version | [7.14] |
| vfts120() | Used in solvebug() demo | [11.19] |