TI 89/92+ Tip List 8.0

Compiled by Doug Burkett, dburkett@infinet.com
Created 12 aug 99, revised 16 July 2000

This is a list of useful tips and hints relating to the TI89 and 92+ calculators. The tips are from user experience, the manual, archived programs or any other source. The tips apply both to operating the calculator and programming in TIBasic.

This list is not a buglist, a wishlist or a FAQ. I maintain an 89/92+ wishlist, and a couple good FAQs are already on the web. The focus of a FAQ is, by definition, frequently asked questions. My purpose with this list is to describe techniques which may not be 'frequently asked', but may be useful nonetheless, to some users.

I'm not trying to rewrite the manual, but the 89/92+ are sophisticated calculators, and some features aren't obvious. Also, all users have different levels of experience, so what is obvious to some is not obvious to all.

I typically check these TI discussion groups for tips: 89/92+, Advanced Math and Electrical Engineering. If you see a good tip or solution in another group, please email me. I would also really appreciate it if you would email me with solutions you develop.

While most of the tips here are short, I welcome submissions of any length.

I am conscientious about giving due credit. That can be hard to do with this kind of list. In general I give credit to the first person to submit a particular tip. I appreciate it if you point out that your tip can be found in the manual, or a FAQ somewhere, or you really learned it from someone else.

If a tip doesn't give credit, that just means that I came up with it on my own, not that I am the original inventor.

Ray Kremer and Bhuvanesh Bhatt have generously volunteered to keep this list on their sites:

Ray Kremer's site:	http://tifaq.calc.org/8992tips.pdf
Bhuvanesh Bhatt's site:	http://triton.towson.edu/~bbhatt1/complinks.html

Thanks, Ray and Bhuvanesh!

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 8.0, I have organized the tips into these categories:

[1] *Documentation*: General documentation for various procedures
[2] *Computer Algebra System (CAS):* Using the CAS for symbolic algebra
[3] *Data structures*: Working with matrices, lists and data variables
[4] *Graphing*: Function graphing and operations on the graph screen
[5] *Graph-Link*: Using the TI GraphLink cables and software
[6] *Math:* Mostly numerical mathematics

[7] *TIBasic programming*: programming techniques

[8] *String*: Manipulating string variables

[9] *User interface*: Handling user input, and displaying output to the user. Includes dialog boxes and results formatting.

[10] *Units*: Using the built-in units conversions

[11] *Solve*: Using the various solving functions, including solve(), nsolve(), csolve(), zeros() and czeros().

I hope you find this useful or interesting, and I am looking forward to support from the TI community.

Doug Burkett
Eaton, OH USA
dburkett@infinet.com

Contributors

The tip author is credited in each tip. I would also like to list them here, and thank them again.

Andy
Kenneth Arnold
Alex Astashyn
Bhuvanesh Bhatt
Billy
Andrew Cacovean (aka Jack_Paper)
Jordan Clifford
Martin Daveluy
George Dorner
Fabrizio
Larry Fasnacht
Glenn E. Fisher
Lars Frederiksen
Mike Grass
Titmité Hassan
Rick Homard
Sam Jordan
Eric Kobrin
Ray Kremer
Christopher Messick
Olivier Miclo
Roberto Perez-Franco
Mike Roberts
Rick A.
TipDS
TM
Frank Westlake
Hank Wu

Revision record: 7.0 to 8.0

- Categorize all tips into topic sections.
- Renumbered all tips
- Developed zip file including code for most tips
- Added Stephen Byrne's list of sites
- Changed description for Frank Westlake's site. TI made him remove the 'ROM images'. Some huge corporations have no sense of humor.
- Added anti-tip #14, base conversions only work in Exact or Auto modes, not Approximate
- Added anti-tip #15, need to use global variables for symbolic calculations in programs
- Changed anti-tip #4, cascading menus. Added note about available work-around.
- Changed [1.2]; noted that combined 89/92+ manual has correct logistic regression equations.
- Changed [2.2]; added solve() method to test expression equality
- Changed [2.14]; added method to create functions in Program editor
- Fixed [3.1], missing a line
- Changed [3.3] from *Delete matrix rows* to *Delete matrix rows and columns*; added method to delete columns, also fixed a typo.
- Changed title of [3.10]; there is no append() function
- Changed [6.4] to Gamma and log-gamma functions
- Added [2.16] Find partial fractions with expand()
- Added [2.17] Use a 'clean' or empty folder for CAS operations
- Added [2.18] Delay evaluation to simplify expressions
- Added [2.19] Restrict function arguments to integers
- Added [3.12] Don't use delta-list function in data editor
- Added [3.13] Find matrix minors and adjoint
- Added [4.8] Plot functions that return lists
- Added [4.9] Faster plots of slow functions
- Added [5.2] Backing up and restoring individual variables may be safer that Get Backup
- Added [5.3] Do not restore a HW1 backup to a HW2 calculator
- Added [6.22] Linear Interpolation
- Added [6.23] Step-by-step programs
- Added [6.24] Fast Fibonacci Numbers
- Added [6.25] Polar and rectangular coordinate conversions
- Added [6.26] Accurate numerical derivatives
- Added [6.27] Find Bernoulli numbers and polynomials
- Added [7.23] Passing user function names as program/function arguments
- Added [7.24] Use a script for self-deleting set-up programs
- Added [7.25] Use scripts to test programs and functions
- Added [7.26] dim() functions slow down with lists and matrices with large elements
- Added [7.27] getmode() returns strings in capital letters
- Added [9.12] Access all custom menus from any custom menu
- Added [10.3] Remove units from numbers
- Added [10.4] Add units to undefined variables
- Added [10.5] Use tmpcnv() to display temperature conversion equations
- Added [11.8] Try solve() for symbolic system solutions
- Added [11.9] nSolve() may return "Questionable Accuracy" warning even with good solutions
- Changed URL for Roberto Perez-Franco's web site.

- Added Stuart Dawson's surveying software site.

1.0 Documentation 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 http://www.ti.com/calc/docs/faq.htm 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 250 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] TI92+ 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 newer combined manual for the 89 and 92+.

[1.3] Alphabetic list of reserved variable names

The 89/92+ have several system variables and reserved names. These can be found in the manual. However, the 92+ module manual list was not updated. The most up-to-date list of reserved names is found in the 89 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 89/92+ Reserved Names

Δtbl	medx2	tc	zeyeθ
Δx	medx3	tmax	zeyeϕ
Δy	medy1	tmin	zeyeψ
Σx	medy2	tplot	zfact
Σx²	medy3	tstep	zmax
Σxy	minX	u1(n) – u99(n)	zmin
Σy	minY	ui1 – ui99	znmax
Σy²	nc	x̄	znmin
σx	ncontour	xc	zplstep
σy	ncurves	xfact	zplstrt
θc	nmax	xgrid	zscl
θmax	nmin	xmax	zt0de
θmin	nStat	xmin	ztmax
θstep	ok	xres	ztmaxde
c1 – c99	plotStep	xscl	ztmin
corr	plotStrt	xt1(t) – xt1(99)	ztplotde
diftol	q1	ȳ	ztstep
dtime	q3	y1'(t) – y99'(t)	ztstepde
eqn	R²	y1(x) – y99(x)	zxgrid
errornum	r1(θ) – r99(θ)	yc	zxmax
Estep	rc	yfact	zxmin
exp	regCoef	ygrid	zxres
eyeθ	regEq(x)	yi1 – yi99	zxscl
eyeϕ	seed1	ymax	zygrid
eyeψ	seed2	ymin	zymax
fldpic	Sx	yscl	zymin
fldres	Sy	yt1(t) – yt99(t)	zyscl
main	sysData	z0max	zzmax
maxX	sysMath	z0min	zzmin
maxY	t0	z0step	zzscl
medStat	tblInput	z1(x,y) – z99(x,y)	
medx1	tblstart	zc	

[1.4] Importing and exporting PC data with the 89/92+

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:

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.

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 89/92+. 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 89/92+, it is faster and easier to create them on the PC, and convert them to the 89/92+ 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 89/92+ 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 89/92+ with GraphLink.
4. Crop the picture on the 89/92+ if needed.
5. Upload the modified picture to the PC, using GraphLink, so you have a backup copy.
6. Archive the picture variable on the 89/92+, if desired, to free some RAM.

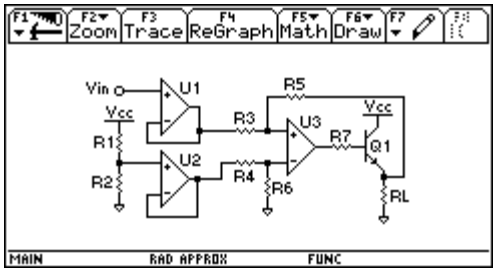
There are a few things you can do to make the picture creation process more efficient:

1. In the PC paint program, set the color depth to "black and white" or "monochrome". The 89/92+ 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.

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

```
ttt(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 leaves 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 a 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 the that two expressions are equal.

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

[2.3] Use when() for absolute values in integrals

AMS 2.03 supports 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)≥0,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+.05*x/n)^n,n,∞)

limit((1+x/(20*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 the version 2.03 of the AMS, you can now use the when function in integration, for example,

```
∫(when(x<1,0,1)x,0,2)
```

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 only 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* error 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,0,∞)
```

If the mode is set to rectangular or real, undef is returned. If we constrain the solution for s>0, like this:

```
∫(t^2*e^(-s*t),t,0,∞)|s>0
```

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 92 w/Plus 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. This can cause unexpected results. For example,

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

does not simplify to zero. Further, the CAS will often convert powers of 1/2 to the square root operator, like this:

$$\left((x)^3\right)^{\frac{1}{2}} \quad \text{simplifies to} \quad \sqrt{x^3}$$

which would be all right if the CAS subsequently recognized that the square root is the same as the 1/2 power. Further, this behavior is not consistent, because

$$\left((x)^{\frac{1}{2}}\right)^3 \quad \text{simplifies to} \quad x^{\frac{3}{2}} \quad \text{not} \quad \left(\sqrt{x}\right)^3$$

This is just something to keep in mind, especially if you write programs that manipulate algebraic expressions.

[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)` returns '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, then use the functions on that variable. Like this:

`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

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

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

Sometimes the 89/92+ CAS cannot find a limit to an expression, because of the way the expression is structured. And, sometimes, the comDenom() function 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, you *can* 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=0,1,sin(x)/x),x,0)
```

If the limit is expressed as

```
limit(when(X=k,L,f(X)),X,0)
```

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=0,1,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. Peform the CAS functions as needed.

[2.18] Delay evaluation to simplify expressions

Sometimes the CAS will not simplify expressions because if 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
```
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 constratints. *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·@n0)
```

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)

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. For example, given 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

```
seq(list[i],i,dim(list),1,-1)
```

to reverse the elements in *list*. For example, if *list* = {1,2,3,4,5}, then the expression 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

```
[[0,∠R▸Pθ(0,0)]]
```

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

$$\angle R\blacktriangleright P\theta(0,0)$$

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],1,0),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 orginal 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 LIST's 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 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 delta-list function in data editor

In the Data/Matrix Editor, you can define functions in the header to create columns of entries. Refer to the 89/92+ manual, p248, for details. AMS 2.03 has a new function, `Δlist`; see manual p463.

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

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

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

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

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() calls 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][0,1,2][0,0,0]])
```

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

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

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 plots functions, so you can't really plot a vertical line. You can, however, approximate a vertical line by defining a function like this:

```
y1=1E100*(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'.

A variation on this theme is to define B as global variable _vert, like this:

```
1E100→_vert
```

then you can define various vertical lines in the Y=Editor:

```
y1={function to be plotted}
y2=_vert(x-0)
y3=_vert(x-3)
```

This will plot the function, and vertical lines at x=0 and x=3

(credit declined)

[4.2] Use DispG to force update to delta-x and delta-y

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 can be demonstrated by this test program:

```
test()
Prgm

clrio

0→xmin:50→xmax
0→ymin:50→ymax

disp "1. ΔX="&string(ΔX)
disp "   ΔY="&string(ΔY)

dispg

disp "2. ΔX="&string(ΔX)
disp "   ΔY="&string(ΔY)
```

```
10→xmin:200→xmax
10→ymin:200→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 the 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*.

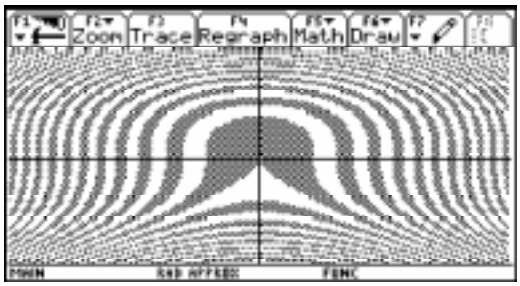
[4.3] Truth plots

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 jack_paper'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)&",τy1,0)"),xx,xmin,xmax,Δx*2),seq(exp
r("when("&string(exp2|x=xx)&",τy2,0)"),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.

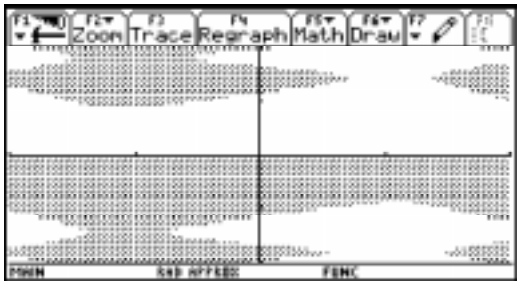
```
truthd(exp1,xxmin,xxmax,yymin,yymax,res)
Prgm
©Truth plot
©Minor change to Jack_Paper's truth() program
©12jan00/dburkett@infinet.com

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)&",τy1,0)"),xx,xmin,xmax,Δx*res),seq(e
xpr("when("&string(exp2|x=xx)&",τy2,0)"),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 Jack'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 Jack_Paper)

[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) | x>=3 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.

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)

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,-11,30})
```

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+ make usually prevent this type of 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
© 18mar00/dburkett@infinet.com

local a,k,n,xd

rowdim(xyd)→n

seq(k,k,n-1,0,-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 and log-gamma functions

The gamma function is one of many 'special functions' that can 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 89/92+ CAS will occasionally return results including the gamma function, but there is no built-in function to calculate the gamma function. 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>0,(z-1)!,undef)
Local n,x,y
```

```
when(real(z)<0,1-z,z)→x
10-min(floor(abs(x)),10)→n
If  n≠0:x+n→x
approx(e^(-x)*x^(x-0.5)*√(2*π)*polyEval({0.000839498720872Ø9,-5.1717Ø9Ø826Ø6E-Ø
Ø5,-0.ØØØ59216643735369,6.9728137583659E-ØØ5,0.ØØØ784Ø3922172ØØ7,-0.ØØØ229472Ø93
6214,-0.ØØ26813271604938,0.ØØ34722222222222,0.Ø83333333333333,1},1/x)*when(n=0,1
,Π(1/(x-k),k,1,n)))→y
approx(when(real(z)<0,π/(sin(z*when(sin(π)=0,π,180)))/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 89/92+ 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)=0 and z<1:Return undef
Local n,x,y
when(real(z)<0,1-z,z)→x
10-min(floor(abs(x)),10)→n
If n≠0:x+n→x
approx((x-0.5)*ln(x)-x+0.5*ln(2*π)+polyEval({-0.ØØØ595238Ø952381,0.ØØØ7936507936
5079,-0.ØØ2777777777778,0.Ø83333333333333},x^(-2))/x+when(n=0,0,ln(Π(1/(x-k),k,
1,n))))→y
approx(when(real(z)<0,ln(π/(sin(z*when(sin(π)=0,π,180))))-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.*

(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
©13mar00/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,"E")-1)),n),"f"&string(exact(n-1)))&rig
ht(s,dim(s)-instring(s,"E")+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)
©14mar00/dburkett@infinet.com
local s1,s2,s3

sum(xl^2)→s3

if h≠0 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)/s3,0}

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,0)
```

To force the fit curve through the origin (0,0), use

```
linreghk(xl,yl,0,0)
```

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=0 and d=0
```

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=0 and d=0 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 number 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≠0 then
 right(str,dim(str)-b)→t
 Return s*(exact(iPart(expr(str)))+exact(expr(t))/10^(dim(t)))

else
 return exact(expr(str))
EndIf

EndFunc
```

The argument is passed as a string, for example

```
stoexact("1.234567890987654321")
```

returns 1234567890987654321/10000000000000000000.

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+ do not always return exact solutions to cubic and quartic equations, even when these equations have solutions. The routines, cubic(), zkubic() and quartic() can be used to get the exact solutions.

cubic():

```
cubic(pp)
Func
Local  pp,qq,aa,bb,cc,dd
pp|z=x→pp
cZeros(pp,x)→qq
If inString(string(qq),".")=0 Then
qq
Else
pp|x=0→dd
d(pp,x)|x=0→cc
(d(pp,x,2)|x=0)/2→bb
(d(pp,x,3)|x=0)/6→aa
zkubic({aa,bb,cc,dd})
EndIf
EndFunc
```

zkubic():

```
zkubic(ù)
Func
Local p,q,r,a,b,u,v,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)="EXPR" Then
 w-a/(3*w)→u
```

```
 w+a/(3*w)→v
 Goto einde
EndIf

If w=0 Then
 If p=0 Then
  -1→u
  1→v
 Else
  0→u
  0→v
 EndIf
Else
 w-a/(3*w)→u
 w+a/(3*w)→v
EndIf
Lbl einde

{u-p/3,-u/2+√(3)/2*i*v-p/3,-u/2-√(3)/2*i*v-p/3}
EndFunc
```

quartic():

```
quartic(pp)
Func
Local qq,ù,ú,a,b,c,d,p,q,r,i,j,k,aa,bb,cc,dd,ee
pp|z=x→pp
cZeros(pp,x)→qq
If inString(string(qq),".")=0 Then
qq
Else
pp|x=0→ee
d(pp,x)|x=0→dd
(d(pp,x,2)|x=0)/2→cc
(d(pp,x,3)|x=0)/6→bb
(d(pp,x,4)|x=0)/24→aa
{aa,bb,cc,dd,ee}→ù
ù[2]/(ù[1])→a
ù[3]/(ù[1])→b
ù[4]/(ù[1])→c
ù[5]/(ù[1])→d
b-3*a^2/8→p
a^3/8-a*b/2+c→q
-3*a^4/256+a^2*b/16-a*c/4+d→r

{1,p/2,(p^2-4*r)/16,-q^2/64}→ú
zkubic(ú)→ù
√(ù)→ù
0→c
For i,-1,1,2
 For j,-1,1,2
  For k,-1,1,2
c+1→c
i*ù[1]+j*ù[2]+k*ù[3]→ú[c]
  EndFor
 EndFor
EndFor

0→k
For i,1,dim(ú)
approx(ú[i])→r
round(real(r),6)→p
round(imag(r),6)→q
For j,1,4
```

```
If p=round(real(qq[j]),6) and q=round(imag(qq[j]),6) Then
k+1→k
ú[i]→ù[k]
EndIf
EndFor
EndFor

ù

EndIf
EndFunc
```

cubic() is used to solve cubic (third-order) equations, and quartic() solves 4th-order equations. Both cubic() and quartic() call zkubic().

All of these routines must be in the same folder. 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^3-12x^2-15x+26)	returns	{-2 1 13}
quartic(z^4-2z^3-13z^2+38z-24)	returns	{-4 1 2 3}

The advantage to 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)

returns these three solutions 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). 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.

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.

(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(1.23456789E10,"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 Jack_Paper)

[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 TIbasic 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
30→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'
0→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: "abs" means absolute error "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

$$\text{absolute error} = |y - y_a| \qquad \text{relative error} = \left|\frac{y - y_a}{y}\right|$$

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 #55. 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 #55, 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 #55.

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 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

$$RMS = \sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(x_i^2\right)}$$

The 89/92+ norm() function is very close to this:

$$norm([a,b,c,...]) = \sqrt{a^2+b^2+c^2+...}$$

so we can rewrite the RMS equation and use the norm() function to find RMS:

$$RMS = \frac{1}{n^2}[norm(dmatrix)]$$

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))^2
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 θ.

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 others 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*57.295779513082→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+ 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,

$$1/(\sqrt{(2\cdot\pi)})\cdot\int(e\text{^}(-t\text{^}2/2),t,1,\infty) = 0.1586\ 5525\ 3456\ 96$$

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 $\int$ 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,0,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, TIBasic 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,-6,0,-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 site: http://perso.wanadoo.fr/ti92-ti89.miclo/frameus.htm

- 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:

$$Fn = \frac{1}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right]$$

However, the 89/92+ cannot use this formula for large n, because the CAS cannot simplify the result. Finding the numbers with a TIBasic 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][1,0]]^(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 θ 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 come up in 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-innDeriv() 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 = π/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) at π/2.01, the call is

nder2("tan",π/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
1E900→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)))))/10→hh
 if hh=0: .01→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",1.5707,"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",1.5707,1E-5) = {1.07771965667E8, 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:

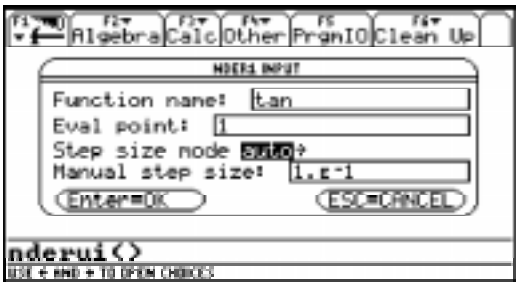
```
nderui()
Prgm
© User interface for nder1()
© 3jan00/dburkett@infinet.com
© Result also saved in nderout
local fn1,xx,steps,smode,reslist,ssz
```

```
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=0: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=0: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()

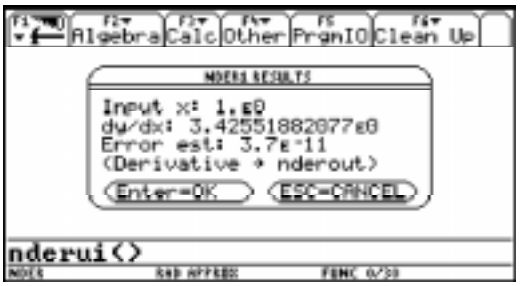
and this input dialog box is shown:



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:



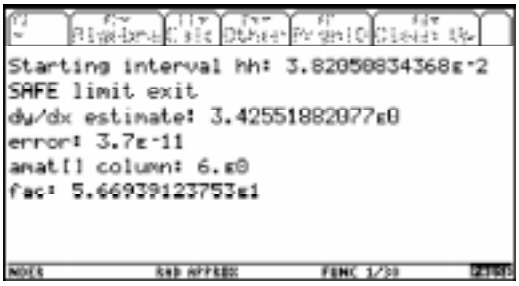
Input x is the point at which the derivative is estimated. *dy/dx* is the derivative estimate. *Error est* is the error estimate. The derivative estimate is saved in the global variable *nderout* in the current folder. Push [ENTER] to find another derivative, or press [ESC] to exit the program.

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 an unacceptably high error.

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:



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"
©6jun00/dburkett@infinet.com
©program version of nder1()

1.4→con
con*con→con2
1E900→big
10→ntab
2→safe

ff&"(xx+hh)"→fphh
ff&"(xx-hh)"→fmhh
ff&"(xx)"→ffun

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)))))/10→hh

 if hh=0: .01→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.

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

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:

$$\zeta(2n) = \frac{(-1)^{n-1}B_{2n}(2\pi)^{2n}}{2(2n)!}$$

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

$B_0 = 1$ $B_1 = -1/2$ $B_2 = 1/6$ $B_4 = -1/30$ B6 = 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
©21jun00/dburkett@infinet.com

if n=0:return 1
if n=1:return -1/2
if n<0:return undef
if fpart(n/2)≠0:return 0

(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 nth Bernoulli polynomial requires finding all the (n-1) previous polynomials, so this can be 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

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

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.
©1jul00/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,0,1)→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
©1jul00/dburkett@infinet.com
©Generates "Domain error" if kk>dim(bpoly)

if kk<0:return "bpo arg error"

when(kk=0,1,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 of large lists, even if those lists do not have many 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.

7.0 TIBasic 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:

$y1=4x^2+3x+2$

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, this is 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.
[0,0]→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=0 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)=0 or dim(b)=0 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 can never 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=0,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 43 or less. When 'k' is 44 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 with the Plus module, AMS version 1.05.

[7.7] Use *return* instead of *stop* in programs for better flexibility, and to avoid a crash

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 TI89/92+ *function* can return a result to the command line, but a *program* usually cannot. However, there is a method to get around this limitation. 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

(Credit to Timité Hassan)

[7.9] Passing optional parameters to functions and programs

Many programming languages support passing optional parameters to functions and programs, but TIBasic 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 programs before archiving for better execution speed

Archiving 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, the 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

TIBasic passes function and program arguments by value, not by reference. For example,

```
myfunc(m1)
```

actually passes the matrix 'matrix1', not the address of 'm1' or a pointer to 'm1' This means that myfunc() cannot write results to variable m1.

To bypass this limitation, pass argument variable names as strings:

```
myfunc("m1")
```

Use indirection within myfunc() to access the variable:

```
myfunc(mat1)
func
```

```
...
#mat1→k
...
k→#mat1
...
endfunc
```

[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)="0-0":return 1.0
if mid(tmp,4,3)="0-1":return 1.01
if mid(tmp,4,3)="1-3":return 1.05
```

```
return 0
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+ provides a few functions which can be used to accomplish some functions 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 faster and uses less ROM.

[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.

[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.

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' 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".

```
setfold(#fname)
```

[7.19] Bypass programs locked with ans(1) and 4→errornum:passerr

It is possible to prevent viewing TIBasic 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 in a 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<0,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
-10→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
10→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<0 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
10→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>0 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 TIbasic 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:

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 p496 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
```

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>0
 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 2 quote characters for each quote you want to create.

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 (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. 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 user'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 10*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 10*lcdrow to 9*lcdrow, but I don't like the way this looks.

[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":0->x
```

This statement is executed before the string() function. The variable is initialized if it doesn't exist, otherwise it is unchanged.

[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

"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.

(Credit to Glenn Fisher, submitted by Larry Fasnacht)

[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"

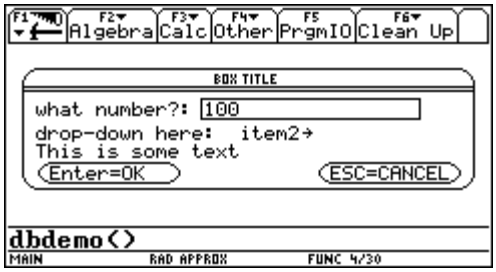
results in a dialog box like this:

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.



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 titletext: string to use for title
Add Request	dbreq(db,promptn,vname)	db: dialog box string promptn: prompt string vname: variable name as a string
Add drop-down menu	dbdrd(db,prmpt,ddlist,ddvar)	db: dialog box string prmpt: prompt string ddlist: list of menu item strings ddvar: variable name as a string
Add text	dbtxt(db,txt)	db: dialog box string 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 4 Text functions and 3 DropDown functions, you can have as many as 2 Request functions.

Number of Text lines ⇨ Number of DropDown lines⇩	0	1	2	3	4	5	6	7	8	9	10
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 • The Title string may be empty; Title ""
Text	• Maximum length of Text string is 30 characters. • If a Dialog box contains Text functions, at least one must contain text.
DropDown STR,LIST,VAR	• STR may be empty. • 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. • LIST may not have any elements wider than 30 characters. • LIST may have a single empty string. • LIST may not have more than 2000 elements.
Request STR,VAR	• STR may have zero characters • STR can be up to 20 characters, maximum.

(Credit to Frank Westlake)

[9.10] 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.

[9.11] 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 E, if any
instring(n,".")→dp
instring(n,"E")→El

©Get exponent or set end of string
if El≠0 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.

[9.12] Access all custom menus from any custom menu

Custom menus are useful for frequently performed operations. Refer to the TI89/92+ User's Guide, p303 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)

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

```
4320.0000000002*(_dollars/_summer)
```

Note that exact(ans()) still returns this answer. If this calculation is performed in EXACT mode, it returns

```
4320.*(_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)	Temperature, degrees Kelvin (K)
Mass, kilogram (kg)	Luminous intensity, candela (cd)
Time, second (s)	Amount of substance, mole (mol)
Electric current, amp (A)	

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(0_°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

```
0.5556·(TF-32)·_°C
```

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:

$$\text{zeros}\left(\frac{x^2-xy+1}{x}, x\right) \mid |y| < 1$$

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=0
```

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_=0})
```

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])&"=0 and "→s
endfor
s&string(eqlist[vdim])&"=0"→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=0: 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 #62 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, the advantage of using this function *does not* come from solving equations by hand, 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 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>0 to find the x = 2 root

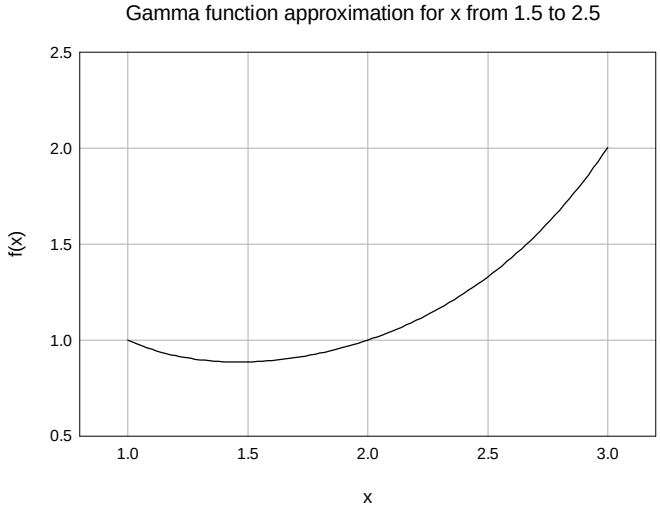
or

nsolve(x^2=4,x)|x<0 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 HP48G 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=10
```

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=10
```

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 TIBasic interprets the solve() result as a boolean, evaluates the boolean and saves the result, 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=0,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(1000*(vf-.015))→vfs
polyeval({3.618919682914,-31.003830334444,76.737472978603,-68.237201523917,262.4
6139741751,84.916629306139},vfs)+polyeval({-3.9287348339733E-7,5.9179552041553E-
5,-0.0036896155610467,0.12308990642018,-2.7560332337098,0},1/vfs)→tapx

©Find solution, with bounds from tapx
nsolve(vfts120(t)=vf,t)|t≥tapx-.4 and t≤min({tapx+.4,705.47})

Endfunc


vfts120(ts)
func

polyeval({6.177102005557E-14,-3.724921779392E-11,9.3224938547231E-9,-1.239459227
407E-6,9.2348545475962E-5,-.003654240052O554,.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 use set 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 TIcares 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
xmax = 100.17 + 1.2E-7
xscl = 1.E-8
ymin = -4.E-13
ymax = 4.E-13
yscl = 1.E-14
xres = 1.

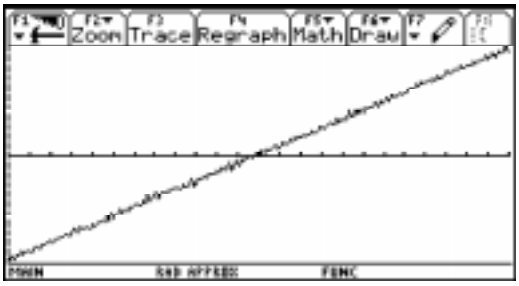
Graph y1(x). The graph shows a greal deal of roundoff noise due to catastrophic cancelation. This roundoff noise can also be seen by generating the Table corresponding to the Graph.

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.

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 89/92+ or TIBasic 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 89/92+ 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.
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 89/92+ 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 [9.11] 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.
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.

More resources - FAQs

This is Ray Kremer's ultimate TI calculator FAQ:

http://tifaq.calc.org/

Texas Instruments maintains four FAQs for the TI89, TI92 and TI92 Plus calculators, and the GraphLink. Note that the TI92 FAQ has a lot more information that either the 89 or the 92+ FAQ, and almost all of it applies to all three calculators.

This is Texas Instruments' FAQ for the TI89: http://www.ti.com/calc/docs/faq89main.htm
It includes these topics:

Year 2000 Information.
Why did TI produce the TI-89?
What is the TI-89?
What is Advanced Mathematics Software?
What do Flash and “electronically upgradable” mean?
How much memory does the TI-89 have?
Is the TI-89 compatible with CBL™ and CBR™?
Will the TI-89 work with the TI-GRAPH LINK™?
Is the TI-89 compatible with the TI-92 and TI-92 Plus?
Which ViewScreen™ LCD does the TI-89 use?
How much will the TI-89 cost?
When and where will the TI-89 be available?
Can I try out the TI-89 with Advanced Mathematics Software?
Processor chip and speed.
Size of the TI-89.
Is the TI-89 approved for College Board tests (SAT, AP, etc)?
Will there be a TI-89 Plus?

This is the FAQ for the original TI92: http://www.ti.com/calc/docs/faq92main.htm
It includes these topics:

Year 2000 Information.
Adding features to the existing TI-92.
Answer in back of the book is different from the TI-92 - why?
Anti-derivatives - why can't the TI-92 find them for all expressions?
arcLen() (arc length) function algorithm.
"Asymptotes" are not drawn when graphing certain functions - why?
AUTO modes function - what is it?
avgRC() (average rate of change) algorithm.
Books and videos.
Cabri Geometry II features not in the TI-92.
Calculator seems to slow down when I insert the link cable.
Calculator will not accept a key press - why?
Calculators that work with the TI-GRAPH LINK.
Causes for memory on Graphing Calculators to clear.
Can I use the link port for my own applications?
Can't see displays because of the classroom lighting.
cFactor() (complex factor) function algorithm.
Changing the order of steps gives different answers sometimes - why?
Circular Definition Error.
comDenom() (common denominator) function algorithm.
Complex numbers sometimes round a component to zero - why?
Construct an ellipse in TI-92 Geometry.
Control pad - how many directions will it move the cursor?
cSolve() (complex solve) function algorithm.
cZeros() (complex zeros) function algorithm.
Define a function from a program prompt.
DERIVE - does the TI-92 have ALL of DERIVE's features?
Determinant - why is it different on different calculators?
Differences in features between the TI-83 and TI-82
Differences between the TI-82 and TI-92.
Differences between the TI-85 and TI-92.
Difference between [diamond] [OFF] and [2nd] [OFF].
Differential equations - why doesn't the TI-92 have them?
Difficulty sending and receiving files with Graph Link and TI-83, 86, and 92.
Dimension of the screen in lines and pixels.
Display - does it scratch?
Display an angle in DMS format on the Program I/O screen.
Division by zero - how can I create this error?
Does TI supply a TI-92 ViewScreen and the separate parts?
d() (symbolic differentiate) algorithm.
Equation for the graph on the cover of the TI-92 manual.
Equation Solver on the TI-92?
Error FOLDER when I try to copy a var in VAR-LINK why?
expand() function algorithm.
factor() function algorithm.
fMax() (symbolic maximum) function algorithm.
fMin(expresn, x) and fMax(expresn, x) return only x values - why?
fmin and fmax - how do they work?
fMin() (symbolic minimum) function algorithm.
fMin() and fMax() are hard to use in programs.
For loop slower on the TI-92 than the TI-8x products - why?
Function counts on graphing calculators.
Gamma Function.
Geometry figure on TI-92 manual - how to create it.

Store a value to a matrix element.
Summation function (symbolic) algorithm?
Superscripted numbers appear on multiple output statements - why?
Symbolic Manipulation (Computer Algebra) - what is it?
taylor() (taylor series) function algorithm.
tCollect() (trig collect) function algorithm.
tExpand() (trig expand) function algorithm.
Trig functions - how do TI products calculate them?
Trig functions return unexpected answers in DEG mode - why?
Verifying a symbolic result.
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 4x get simplified into 22x?
Why, when I enter solve (sin(x)=0,x) do I get x=@n1*pi? (TI-92).
zeros() function algorithm.

This is the FAQ for the TI92+: http://www.ti.com/calc/docs/faq92pmain.htm
It includes these topics:

Year 2000 Information.
What is the TI-92 Plus Module?
Why did TI produce the TI-92 Plus Module?
How much more memory does the TI-92 Plus Module have?
What is Advanced Mathematics Software?
What do Flash and “electronically upgradable” mean?
When will the TI-92 Plus Module be available?
How much does the TI-92 Plus Module cost?
Where do I buy the TI-92 Plus Module?
Can I buy the TI-92 Plus Module already installed in a TI-92?
Does the TI-92 Plus do everything that the TI-92 does?
What are the main differences between the TI-92 and TI-92 Plus?
Is the TI-92 Plus compatible with the CBL™/CBR™?
Will the TI-92 Plus use the same ViewScreen™?
Will it work with the TI-GRAPH LINK™?
Are there currently any software upgrades available for the TI-92 Plus Module?
How much will software upgrades cost?
What are the appropriate courses for the TI-92 Plus?
Are there any support materials or workbooks for the Advanced Mathematics Software of the TI-92 Plus Module?
Will the TI-92 Plus be allowed on College Board tests (SAT, ACT)?
Are the TI-92 and TI-92 Plus compatible?
What CANNOT be shared between the TI-92 and the Advanced Mathematics Software of the TI-92 Plus Module?
What types of 1st- and 2nd-order ODE's will Advanced Mathematics Software solve symbolically?
What types of Systems of Equations can be solved with Advanced Mathematics Software?
Are there any Engineering applications for the TI-92 Plus platform?
Is user data archive memory erased by reset?
Is ALL memory reset when I insert the module?

This is the GraphLink FAQ: http://www.ti.com/calc/docs/faqgraphlinkmain.htm
It contains these topics:

Year 2000 Information.
Backing up the TI-86 with Graph Link.

Build my own link cable?
Calculator seems to slow down when I insert the link cable.
Calculators that work with the TI-GRAPH LINK.
Can I use a Printer port instead of Modem port on Macintosh?
Difficulty sending and receiving files with Graph Link and TI-83, 86, and 92.
Edit-lock (protect) TI-85 programs with Graph Link?
Extra spaces in front of each program line - why?
File size limitations - why can large files from my calculator not be opened?
Graph Link Crashes when I open a TI-85 program - why?
Graph Link (DOS) interferes with my mouse or other serial card.
I cannot open a "protected" TI-82 program using my Macintosh Graph Link.
Inserting lists into (exporting from) an Excel™ spreadsheet.
Installing Graph Link fonts on my Macintosh.
Is Graph Link copy protected?
Macintosh 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.
TI-GRAPH LINK and TI-80 - how does it work.
Transmission errors with Graph Link (Macintosh version).
Transmission errors with Graph Link (PC version).
Why can't I open an edit-locked program using my Graph Link?

More resources - TI documentation

TI publishes a newsletter for calculator users called *EightySomething!.* These often have good tips and examples. You'll need Acrobat Reader, as the files are in .PDF format. Get the current and past issues here:

http://www.ti.com/calc/docs/80xthing.htm

TI calculator manuals, called guidebooks, are available on-line. These are the links to download the manuals:

TI92 guidebook:	http://www.ti.com/calc/docs/92guide.htm
TI92 Plus module guidebook:	http://www.ti.com/calc/docs/92pguide.htm
TI89 guidebook:	http://www.ti.com/calc/docs/89guide.htm
Combined 89/92+, AMS 2.03	http://www.ti.com/calc/docs/8992pguide.htm
TI GraphLink guidebook:	http://www.ti.com/calc/docs/linkguide.htm

More resources - web sites

There are thousands of independent TI calculator web sites. These are the best. In my opinion, of course.

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 TI89/92+ site: http://perso.wanadoo.fr/ti92-ti89.miclo/frameus.htm

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://symbulator.i.am/

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 89/92+.

S.L. Hollis' TI89/92 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.

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 TI89 and the HP49G, with little or no bias. Some math programs. Good assembly programming tutorials.

SoftWarehouse advanced functions: http://www.ti.com/calc/docs/swh/library.htm

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 calculator program archives: http://www.ti.com/calc/docs/arch.htm

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/complinks.html

Several TI92+ 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. 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.