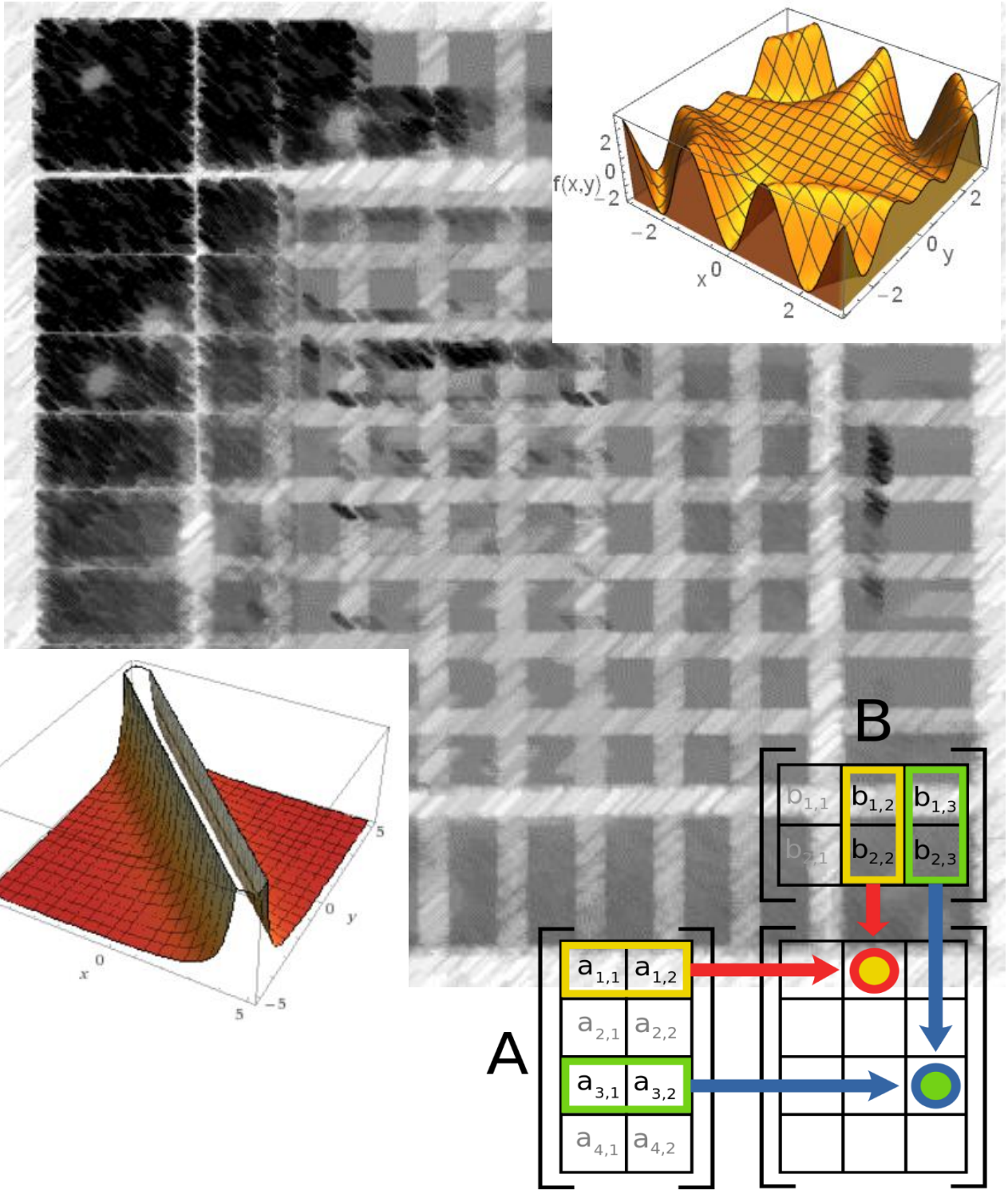
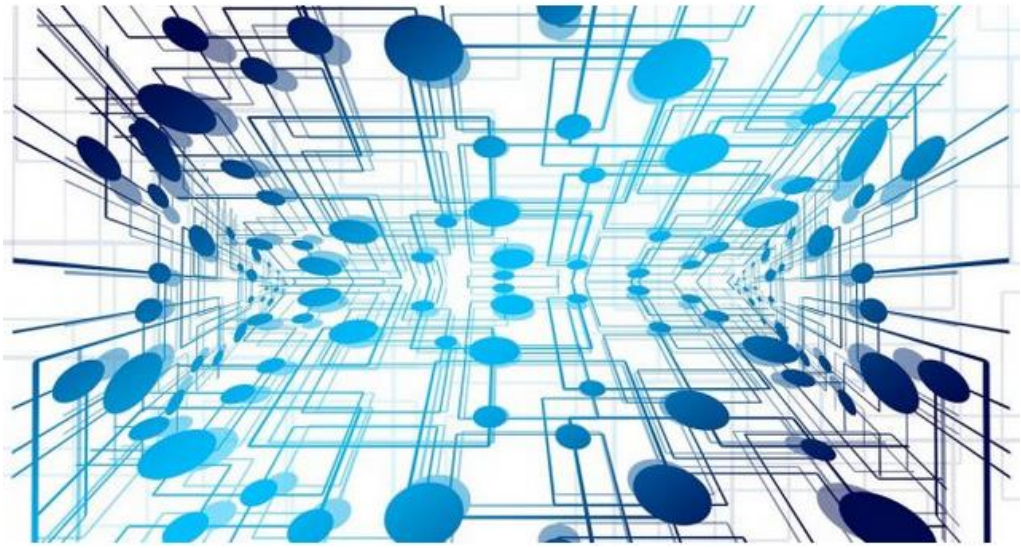
HP-41 Advantage Math ROM

Extending the HP-41 SandMatrix - I



Ángel M. Martin Cañas. *February 2020*

This compilation revision 1.3.3

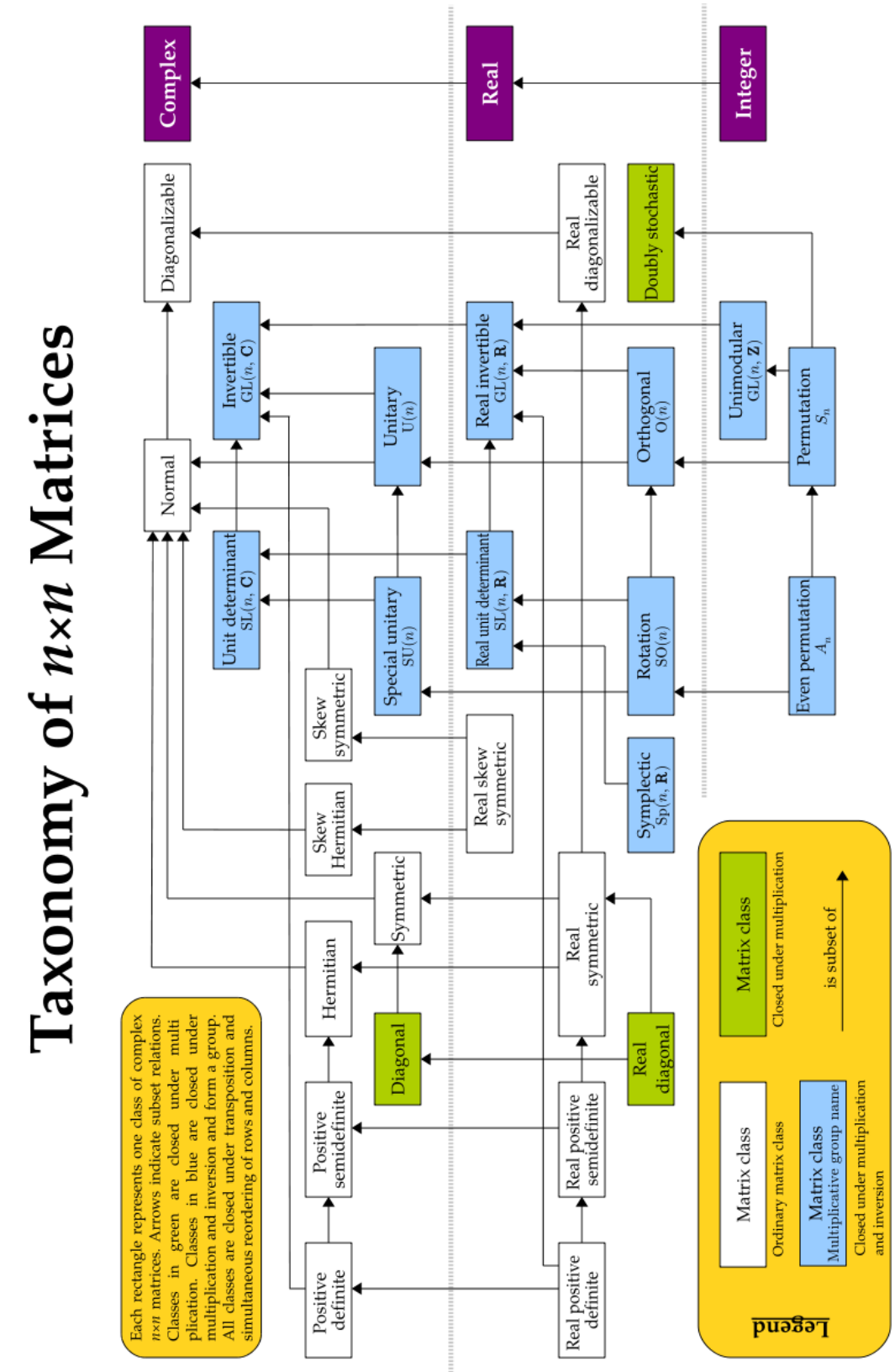


Front cover image taken from: *https://www.dreamstime.com/royalty-free-stock-photography-mathematics-background-image20849947*

Screen captures taken from V41, Windows-based emulator developed by Warren Furlow.
See www.hp41.org

ADVANTAGE_MATH 2C
HP-41 Module

Table of Contents


Taxonomy of n×n Matrices
Complex
Real
Integer
Diagonalizable
Real diagonalizable
Doubly stochastic
Invertible GL(n, C)
Real invertible GL(n, R)
Unimodular GL(n, Z)
Normal
Unitary U(n)
Orthogonal O(n)
Permutation Sn
Unit determinant SL(n, C)
Real unit determinant SL(n, R)
Special unitary SU(n)
Rotation SO(n)
Even permutation An
Skew symmetric
Real skew symmetric
Skew Hermitian
Symplectic Sp(n, R)
Hermitian
Symmetric
Real symmetric
Positive semidefinite
Diagonal
Real positive semidefinite
Real diagonal
Positive definite
Real positive definite
Each rectangle represents one class of complex n×n matrices. Arrows indicate subset relations. Classes in green are closed under multiplication. Classes in blue are closed under multiplication and inversion and form a group. All classes are closed under transposition and simultaneous reordering of rows and columns.
Legend
Matrix class
Ordinary matrix class
Matrix class
Multiplicative group name
Closed under multiplication and inversion
Matrix class
Closed under multiplication
is subset of

Introduction: Bringing it all home.

Much (and mostly good) has been said about the HP-41 Advantage Pack , released by HP in what could arguably be already the tailing end of the HP-41's life – even if that was self-inflicted, due to the further introduction of subsequent models claiming to be its successor (such as the HP-42S and the HP-48). But in retrospect one would notice the apparent lack of programs written taking advantage of the powerful platform provided by the Advantage Pac: those are really far and in between in the literature, and one has to do a serious research effort to find the few ones available, now incorporated to this module.

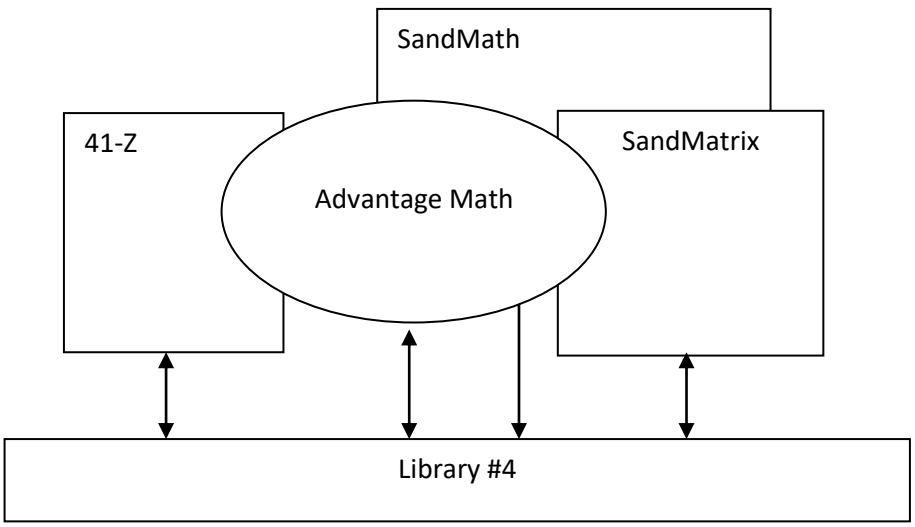
Surely power-house functions like **MSYS** and **MDET** cannot be improved upon, but there are many areas where they could be put to work towards more ambitious goals, like Cubic Spline interpolation and systems of Non-Linear equations – both included in the module as well.

Obviously the SandMatrix already brought the state of the art to a further place, but this module picks up where the SandMatrix left off, mainly adding some sorely missing routines to handle complex matrices in a more convenient way and extending the built-in capabilities of the **FROOT** and **FINTG** facilities. It also "brings it all home" with a few applications that use functions from several other math modules together: the 41Z, SandMath and SandMatrix all together at unison, not only for the complex matrices section but also in unexpected places such as real function derivatives using the Complex Step technique, or the Bessel functions calculated via continued fractions… quite something to behold.

Module Dependencies.

As mentioned, you should have the 41Z, the SandMath and the SandMatrix plugged in the calculator. They will use 6 pages of the I/O bus, to add to the single page required by this ROM. This means that only one page will remain available in the external ports of the calculator, so use it judiciously – I strongly suggest the OS/X Module to be plugged as well, and if possible (i.e. without a printer) in page #6. The WARP Module in page#7 (or the Power-CL for CL owners) will round up the perfect set.

It comes without saying that the Library#4 is needed as well, as a pre-requisite for all the modules mentioned before. And lest we forget, the HP-41 CX is required (the X-Functions won't cut it, sorry).



ROM Function Tables.

Without further ado, let's see the functions included in the module. Refer to the individual function descriptions later on for details on the syntax and use instructions.

XROM	Function	Description	Input	Author
12,00	**-ADVTG_MATH**	*Section Header*	*n/a*	
12,01	**"AIM"**	Anti-Identity Matrix	order in X	*Ángel Martin*
12,02	**"CSORT"**	Column Data Sort	*Using Matrix File*	*Simon Bradshaw*
12,03	**"DDN"**	Harmonic Determinant	order n in X	*Ángel Martin*
12,04	**"DN"**	Verification formula	order n in X	*Ángel Martin*
12,05	**"FCTORS"**	Builds Factors Mtrix	Name in ALPHA	*Ángel Martin*
12,06	**"LS2"**	Least Squares 2nd order	*Parabolic Regression*	*David Hodges*
12,07	**"M90L"**	Rotates 90-deg Left	Name in ALPHA	*Ángel Martin*
12,08	**"M90L"**	Rotates 90-deg Right	Name in ALPHA	*Ángel Martin*
12,09	**"MMIRR"**	Mirror Image	Name in ALPHA	*Ángel Martin*
12,10	**"MCSRT"**	Matrix Column Sort	Col# in X, Name in ALPHA	*Richard Kendon*
12,11	**"MINOR"**	Matrix Principal Minor	I,j in X , Name in ALPHA	*Ángel Martin*
12,12	**"MINORS"**	Builds Minors matrix	Name in ALPHA	*Ángel Martin*
12,13	**"ML"**	Matrix Least-Squares		*David Hodges*
12,14	**"MSRT"**	Matrix Sort	*for Data registers*	*John Bruce Jr.*
12,15	**"MRREV"**	Matrix Row Reversal	Name in ALPHA	*Richard Kendon*
116,16	**"MSYM"**	Matrix Symmetric	Name in ALPHA	*Ángel Martin*
12,17	**"MU<>L"**	Swaps U/L Regions	Name in ALPHA	*Ángel Martin*
12,18	**"MZRO"**	Zeroes a Matrix	Name in ALPHA	*Richard Kendon*
12,19	**"OCS"**	Over-Conditioned Systems	*Final Stage*	*David Hodges*
12,20	**"OCS+"**	Over-Conditioned Systems	*Main Program*	*Ángel Martin*
12,21	**"R/aRR"**	Make diagonal Unitary	Name in ALPHA, i.j in X	*Ángel Martin*
12,22	**"R1DP"**	Row-1 duplication	Name in ALPHA	*Richard Kendon*
12,23	**"SUBMAT"**	Reduces Matrix	Name in ALPHA, i.j in X	*Ángel Martin*
12,24	**-2D_ITG/SLV**	*Section Header*	*n/a*	*n/a*
12,25	**"FITG2"**	Double Integrals;	FNAME in ALPHA, limits	*Ángel Martin*
12,26	**"*2D"**	Inner Integral	g = g(x)	*Ángel Martin*
12,27	**"F1XY"**	x+y	Example 1	*Ángel Martin*
12,28	**"F2XY"**	y cos(p xy)	Example 2	*Ángel Martin*
12,29	**"FRT2"**	Double Solver	FNAME in ALPHA, guesses	*Ángel Martin*
12,30	**"*FG"**	Inner Solver	f = f(x)	*Ángel Martin*
12,31	**"F1"**	sin(x + y) = x	Example2a	*Ángel Martin*
12,32	**"F2"**	cos(x - y) = y	Example2b	*Ángel Martin*
12,33	**"G1"**	x^2 + y^2 = 5	Example 3a	*Ángel Martin*
12,34	**"G2"**	x^2 - y^2 = 3	Example 3b	*Ángel Martin*
12,35	**CLOAK**	Hides buffer 14 into #13	none	*Ángel Martin*
12,36	**EXPOSE**	Exposes Buffer 14 from #13	None	*Ángel Martin*

12,37	**RESET**	Resets Buffer 13	None	*Ángel Martin*
12,38	**-INTERPOL**	*Section Header*	*n/a*	*n/a*
12,39	**AINT**	ALPHA integer X	X in X	*Fritz Ferwerda*
12,40	**ASWAP**	ALPHA Swap around comma	A,B in ALPHA	*Ángel Martin*
12,41	**CLAC**	Clear ALPHA from Comma	String in ALPHA	*W&W GmbH*
12,42	**E3/E+**	Builds pointer	x.00X	*Ángel Martin*
12,43	**FLNAME**	Working File Name	None	*Sebastian Toelg*
12,44	**"CSPLINE"**	Cubic Spline	Interpolation	*Greg McClure*
12,45	**"DFED"**	Data File Editor	CF 08: Edit	*Ángel Martin*
12,46	**JACOBI**	Jacobi Method	Under program control	*Valentín Albillo*
12,47	**POLZER**	From Zeros to Roots	Under program control	*Ángel Martin*
12,48	**"JYNX"**	Bessel J & Y via Cont. Fractions	n in Y, x in X	*Martin-Baillard*
12,49	**"="**	Complex Cont. Fraction	under PRGM control	*Martin-Baillard*
12,50	**"#"**	Real Cont. Fraction	under PRGM control	*Martin-Baillard*
12,51	**-NL_SYSTEMS**	*Section Header*	*n/a*	*n/a*
12,52	**"NLSN"**	Non-Linear System Driver	Driver program	*Ángel Martin*
12,53	**"NLSYS"**	Non-linear Systems	*Main Program*	*Greg McClure*
12,54	**"FIN"**	Input Function Names	*n expected in R00*	*Ángel Martin*
12,55	**“PLR+”**	Driver for PLR	Prompts for data	*Ángel Martin*
12,56	**“PLR”**	Polynomial Real Roots		*JM Baillard*
12,57	**"XIN"**	Input guess values	*n expected in R00*	*Ángel Martin*
12,58	**"XOUT"**	Output Results	*n stored in R00*	*Ángel Martin*
12,59	**"XHALL"**	Halley'S Method w/ DERV	h in Y, x0 in X	*Ángel Martin*
12,60	**"XNWT"**	Newton's method w/ DERV	h in Y, x0 in X	*Ángel Martin*
12,61	**"ZNWT"**	Newton's Method w/ 41Z	h in Y, x0 in X	*Ángel Martin*

1. Advantage Pac Applications

Taken from old Data File issues, these few applications illustrate the "discovery" of the matrix functions by the UK community – apparently oblivious to the previous existence of the very interesting (albeit inferior) set provided by the CCD Module… The descriptions are taken directly from the original sources.

Matrix Routines using the Advantage ROM

R.D. Kendon, DataFile V9N7p32

1. Matrix Rows Reversal

With the matrix name in ALPHA, this routine uses **MSWAP** starting with the outer rows and working to the middle. The program makes use of the convention, described in the Advantage manual, than an integer B in X or Z is taken as B.001, whereas in Y is taken as B.00n where n is the last column.

```
1   LBL "RVR"
2   DIM?
3   INT
4   1
5   LBL 01
6   ENTER^
7   MSAWP
8   SIGN
9   ST- Z
10  X<Y?
11  GTO 01
12  END
```

2. Make all matrix elements zero.

When some elements are ALPHA data you cannot place 0 in X and use **MAT***. Use the fact that resizing a matrix to larger dimensions (as opposed to creating a matrix) sets all additional elements to zero. Matrix name in ALPHA.

```
01 LBL "MZ"
02 DIM?
03 0
04 MATDIM
05 MS
06 RDN
07 MATDIM
08 RTN
```

3. Row-1 Replication.

I had a matrix of data and I wished to multiply all elements of each column by a constant which was specific to that column. These constants were in single-row matrix, but one cannot use **MAT*** unless the matrices are of equal dimensions. This program assumes that a multiplier matrix of the appropriate dimensions is made and that the constants are in row-1. These constants are first moved to the last row using **R<>R**.

The program then sets the stack so that all rows except the first row are moved, as a block, up one row. The manual states that where the source and target blocks overlap the function works on the last group upwards, so this gives the desired result.

```
1   LBL "RDP"
2   DIM?
3   INT
4   R<>R
5   SIGN
6   0
7   ENTER^
8   2
9   MMOVE
10  END
```

4. Matrix Rows Sorting.

Use this routine to sort the order of rows of a matrix, using a specified column as the key. It was written about the same time as that by John Bruce in V5N4p23. The matrix must be in Main memory since **ANUM** is used to locate the matrix header register. To sort a matrix of several columns, the key column should be in X (e.g. 2 to sort on second col), and F00 should be clear. Since it is necessary to use CMAXAB the elements of the key column should all be of the same sign. If F00 is set, the X-register is not used, and the sort uses MAX. In this form the program is virtually the same as John Bruce's.

```
01  LBL "CSR"
02  1
03  ALENG
04  -
05  X=0?
06  AIP
07  RDN
08  "|-,"
09  DIM?
10  X<> M
11  STOO
12  RDN
13  STO  N
14  0
15  RCLM
16  LBL 00
17  +
18  MATDIM
19  RCL  N
29  FC? 00
21  CMAXAB
22  FS? 00
23  MAX
24  RDN
25  RDN
26  MRIJ
27  INT
28  RCL  Y
29  E3
30  /
31  +
32  R<>R
33  RDN
34  FRC
35  LAST X
36  INT
37  DSE X
38  GTO 00
39  X<> M
40  MNAME?
41  ANUM
42  X<>Y
43  STO IND Y
44  MATDIM
45  END
```

Sorting with the Advantage ROM - John Bruce Jr. – DataFile V5N4 p23

INTRODUCTION. The two functions MAX and MIN included in the set of instructions with the ADV ROM are very powerful additions to the Sorters "elbow". Not only do they act on numeric data, but they also deal with ALPHA data. This can only be a major breakthrough for 41C/41CV owners.

EXECUTION OF SORT. The program expects to see a defined array in Main memory containing your data(a mixture of numeric and alpha data is allowed).Place the name of the array in ALPHA and execute SORT.SPEED: On average this program is 1/3 faster than Binary Insertion Sorting.

```
01 LBL 'SORT"
02  0
03  MSIJA
04 DIM?
05  RCL X
06  E3
07  /
08  ANUM
09  +
10  X<>Y
11  *LBL01
12  STO IND Y
13  MATDIM
14  MAX
15  RDN
16  RCL X
17  E3
18  /
19  MRIJ
28  INT
21  +
22  R<>R
23  RDN
24  DSE X
25  GTO01
26  STO IND Y
27  X<>Y
28  FRC
29  E3
38  •
31  MATDIM
32 .END.
```

Data sorting using the Advantage ROM matrix functions.

Simon Bradshaw -DataFile V4N5 p22

Requirements: HP-41 (any model), Advantage ROM (XF/M - optional but very helpful)

The reviews that have appeared in DATAFILE - and some of the articles referring to the ADV ROM - have said, with justification, much about its outstanding matrix manipulation functions. These are intended mainly for mathematical work, e.g. solving simultaneous equations or use of other matrix-based problem-solving techniques. But they can be used in less directly 'number-crunching' ways as well.

What is a matrix? To a BASIC programmer a matrix is just a two-dimensional array. What are arrays used for? "Data storage and manipulation" is quite often the answer. This suggests that the matrix functions can be used to allow us to store and work on blocks of data. In fact, when the functions available in the ADV ROM for working on matrices are compared with those usually available to the BASIC programmer for array handling, the 41ADV user seems to have quite a big 'advantage' (pardon!). The matrix 'utility' functions allow exchange of pairs of rows and columns, swapping and moving of parts of a matrix or matrices, comparison of rows (more on which below), and a number of functions which while 'mathematical' in nature are very helpful in data processing, e.g. row and column sums, largest and smallest value checks, and several others. Add to this the option of automatic sequential access to elements and you have a collection of array handling functions which put most BASICs to shame.

Of course, there are disadvantages - you cannot save space by use of integer variables, and so you will always need at least one register per element. Also, each element can only store six characters of ALPHA information, although there is no reason why you could not spread a longer ALPHA string over two or more elements in the same row.

Discussing ALPHA elements poses a few questions. The advantage ROM manual has little to say on the latter (it suffers from this problem throughout), other than that you can enter ALPHA values via the matrix editor and that most functions"*…are not meaningful for matrices containing ALPHA data …*"(p43). In fact, the list of error messages on p38 includes ALPHA DATA and implies that you cannot operate on matrices with ALPHA elements.

In fact, most of the 'utility' functions can operate perfectly happily on ALPHA elements - they only move elements around, and do not operate on them. At first this does not seem to helpful, since to do sorting you have to be able to do string comparison. A function is provided '**R>R?**' which takes an argument kkk.lli in X and compares elements in rows k and l of the matrix, working through columns until it finds two unequal elements and then giving 'true' if the element in k is greater than that in l, and 'false' (and skipping a step in a program) otherwise.

At least that is the manual's explanation - no mention is made of how it handles ALPHA data. I had written a program, given below, to sort an n-row matrix, and decided to see how it would handle ALPHA data. Entering this sort of data value '**MEDIT**' is quite easy - just press ALPHA before entering the data (remember only the first 6characters will be stored into the matrix elements. I entered a series of short names and ran my matrix-sorting program. To my surprise, not only was there no ALPHADATA message flashed at me, but when I checked the matrix, I found the elements sorted into alphabetical order!

Further checks confirmed that R>R? acts like the HP-41CX indirect comparison functions, in that it does a proper alphabetical order test (e.g. giving the result 'HOLE'> 'HOLD'). As far as I know this information is not given elsewhere - the Advantage manual says nothing about ALPHA sorting capability, and when I called Wlodek he knew nothing about it. It would seem that Hewlett-Packard have acted true to form, and the undocumented function has 'struck again (cf HP-75 I/O ROM).

The upshot of all this is that the Advantage can be used to sort alphanumeric lists. Consider the list of the following HPCC members, giving their first names and their membership numbers.

Colum →	1	2
1	DAVID	2
2	WLODEK	9
3	RABIN	19
4	DAVID	155
5	SIHON	398

If this is sorted using my program, then the matrix is rearranged as below:

1	DAVID	2
2	DAVID	155
3	RABIN	19
4	SIMON	398
5	WLODEK	19

The names have now been sorted into alphabetical order. Since each row is moved as a whole, each entry keeps its appropriate membership number. Note how the entries for Messrs Burch and Bundy have been sorted by number. Since the comparison function **R>R?** moves through the columns of the matrix until it finds two unequal elements, on finding the first-column elements both containing 'DAVID', it went on to compare the second-column elements in each row - here the membership numbers - and sorted the rows by them.

The actual sorting routine - **MSORT**- is given below. It is a reasonable approximation to what a true sort function should do in that it preserves ALPHA and the stack (except L, as a true function would) and uses no main memory registers. As it stands, it requires extended functions and memory, but this is not essential, although without them, thestack will not be preserved,

```
01 LBL "MSORT;Matrix Sort
02 SIGN      ;Save X in L
03 RDN       ;Drop stack
04 DIM?      ;make matrix current
05 RDN       :Drop stack
06 4         ;Size of data file
07 "TEMP"    ;Name of data file
08 CRFLD     ;Create temp data file
09 R^        ;Roll stack up
10 SAVEX     ;Save T
11 R^
12 SAVEX     ;Save Z
13 R^
14 SAVEX     ;Save Y
15 LASTX
16 SAVEX     ;Save X
17 MNAME?    ;Restore matrix name
18 DIM?      ;Get matrix dimensions
19 INT       ;Get number of rows
20 LBL 01
21 2.001     ;Index counter
22 CF 00     ;Check-if-sort-made flag
23 LBL 02
24 R>R?      :Compare rows
25 GTO 03    ;If in order, jump on
26 R<>R      ;If not, swap rows
27 SF 00     ;indicates swap made
28 LBL 03
29 1.001     ;Increment index counter
30 +
31 ENTER     ;Duplicate counter
32 FRC       ;Extract number of first
33 E3        ;element in the next
34*          ;row pair to be sorted
35 RCL Z     ;Recall no of rows
36 X=Y?      ;Test if equal
37 GTO 04    ;If so, pass all rows checked
38 RDN       ;If not, get back index counter
39 RDN       ; … and check next pair of
rows
40 GTO 02    ;go to row check
41 LBL 04
42 FS?C 00   ;Were any sorts made?
43 GTO 01    ;if so, then go and sort again
44 "TEMP'    ;Name of data file
45 CLX       ;Set pointer to start …
46 SEEKPTA   ; … of data file
47 GETX      ;Recover T
48 GETX      ;    "    Z
49 GETX      ;    "    Y
50 GETX      ;    "    X
51 PURFL     :Purge data file from XM
52 MNAME?    :Restore matrix name to
APLHA
53 END       ;End of program
```

A few notes on this program. If you don't have XF/M, then just delete lines 02 to 17 and 44 to 52. The program will work just as well, but it will corrupt the stack. The stack save routine is effectively that given in 'Extend Your HP-41', while Wlodek suggested to me the idea of saving the stack in XM when synthetic programming was difficult. Here the APLHA register is needed and so it is hard to use registers M, N and O to store anything in. Although it could probably be done, it would have made the program (a) longer and (b) harder for me to write, and so I usedX-Mem instead.

Notice that I didn't need to save ALPHA. Just by using a dummy function (here **DIM?**) I made the matrix named the current matrix, and so later I could recover ALPHA by use of **MNAME?** This also checks if the matrix named is present before starting to save the stack in XM.

The sorting method used is the bubble sort (or ripple sort), which works by comparing each row with the next to see if they are in the right order, swapping them if they are not. It then loves on one row and repeats the process. Once it has worked through all the rows it checks if any sorts were made. If not, then the "rows" must be in order and it finishes, otherwise it goes back and sorts again. This routine is not very fast but is easy to program. As for speed, to sort a matrix of 6 rows takes this program 35 seconds - this is for a matrix in reverse order, which takes the most sorting, so this is a maximum time. To sort a 10-row matrix takes a maximum of 104 seconds -, I have worked out that the time taken to complete a bubble sort is proportional to n(n-1) where n is the number of rows. This agrees very well with my timings, which seems to indicate that for my program to sort an n-row matrix takes n(n-l)17/6 seconds. To sort a 200-row matrix (say 20013, with one column holding numbers of active HPCC members and the other two their names) should thus take almost 13 hours.

This may appear to be rather a long time but bear in mind that **MSORT** is written in FOCAL (41 User language) and so cannot be expected to be too fast, but since it is written using the M-code routines in the ADV ROM it is still faster and simpler than a program for the 41 alone. By the way, the times given were for matrices in main memory. If your matrix is stored in extended memory (as the one I described above would have to be) timings indicate that about 10% more time is needed to complete a sort of a given matrix. Also, if using XM, remember to leave some space for the data file that MSORT creates.

A couple more notes on matrix sorting. As it stands, **MSORT** sorts in ascending order. Modifying the program to sort a file into descending order is very simple - just replace lines 25 to 28 with:

```
SF 00, R>R?, R<>R
```

This effectively inverts the test so that now a matrix is sorted with the largest element first.Earlier I said that when **R>R?** looks at two rows it goes through them column by column until it finds two that are not equal. This means that you can spread names out over lore than one columns, e.g. consider the following
(unsorted) matrix:-

```
(-------)   (--------)    <- 6 chars each
BRADBU      RY      R
BRADSH      AW      S
BRADSH      AW      D
BLACKB      USH     S
BLACKB      URN     P
```

MSORT sorts this into the following:-

```
BLACKB      URN     P
BLACKB      USH     S
BRADBU      RY      R
BRADSH      AW      D
BRADSH      AW      S
```

Note how BLACKBURN and BLACKBUSH have been sorted into the right order, as have BRADSHAWD and BRADSHAW N. You may think that (say) BRADSHAWN K might confuse the program by being sorted between these two, but because **R>R?** compares characters within a matrix element one character at a time, the "N" of BRADSHAWN is compared with the "_" after BRADSHAW, and since sorting is done by ASCII code "_" comes before "N" and so immediately **R>R?** decides that BRADSHAWN comes after BRADSHAW, whatever initials follow them.

Thus, can be extended to as many rows as you like, within limits of memory, for instance to get my HPCC members list into memory all names would have to be abbreviated to a maximum 11 letters plus one initial (sorry about that, Wlodek!). Incidentally, if you want to sort such a matrix by membership number at one tile and name at another, then just use the matrix manipulation commands to rearrange the column within the matrix, e.g. by using **C<>C** (column exchange) you can bring any column to the 'front' of a matrix and then sort the matrix using that column.

If you have a file as big as this, you will need to save it. If the file is in Main Memory, work out its start and end addresses, and use **WDTAX** or **WRTRX** to save it. If the file is in XM then use **MMOVE** to copy the file to a dummy matrix in Main memory and save that as described. If your XMmatrix is very large, you may need to do this lore than once, saving the XM matrix in two or more parts.

While we are on the subject of moving around our nicely sorted data, you will no doubt want to write programs to help you enter and extract data from matrices such as I have described - MEDIT is rather inconvenient and slow for entering ALPHA data, especially if that data is to be broken up over several columns. It is easier to write a program which reads the data into ALPHA and then uses ASTO X, MSR+ several times. If you write any good programs in this vein, then please send them in. If you write an improved version of MSORT, then please send that in - I am sure that someone out there can write a version in half the number of lines which preserves the stack without using XM. If you actually find a use for **MSORT** - or any other ADV ROM-using data sorting/storage routines, then write to DATAFILE about it - how people use their 41's is always interesting reading.

Parabolic Regression - by David Hodges (DataFile V4N5 p11

System requirements: HP-4l with Advantage ROM

The HP-4l Advantage ROM contains a program called **CFIT** which enables you to fit four different types of curve to a set of statistical data (xi ,yi). Whilst this is ideal for the majority of applications, there are occasions where it is better to fit a second order polynomial to the data. This means expressing y in terms of x like this:-

Y = a0 + a1.x +a2.x^2

and obtaining the best coefficients a0 and a1 using the method of least squares. This technique is also known as "parabolic regression" and there is a program in the Stats ROM which you can use to calculate the coefficients. If you do not have a Stats ROM but have an Advantage ROM, however, you can use the short program below which uses the module's matrix functions. It is written along the same lines as **CFIT** and has a similar menu but is does not have aΣ- facility and does not calculate a correlation coefficient.

LS2 (least squares, 2nd order) works by allowing you to accumulate paired data in the same way as you would with Σ+. When all of the data have been entered the program sets up a system of matrices from the summations and puts the matrix files in extended memory f this is available. The**MSYS** function is then used to solve the system and obtain the coefficient matrix.

$$\begin{bmatrix} \sum x^0 & \sum x^1 & \sum x^2 \\ \sum x^1 & \sum x^2 & \sum x^3 \\ \sum x^2 & \sum x^3 & \sum x^4 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \end{bmatrix} = \begin{bmatrix} \sum y \\ \sum xy \\ \sum x^2 y \end{bmatrix}$$

initialize the routine, clear the summation registers and display the menu. Lines 14 to37 perform the summations and store the values in registers 00 to 07. Registers 0to 10 are used to store a0 , a1 and a2after computation.

r00 .... $\sum x^0$ (n)
r01 .... $\sum x^1$ ($\sum x$)
r02 .... $\sum x^2$
r03 .... $\sum x^3$
r04 .... $\sum x^4$
r05 .... $\sum y$
r06 .... $\sum xy$
r07 .... $\sum x^2 y$
r08 .... a_0
r09 .... a_1
r10 .... a_2

Two matrices are created by lines 38 to 72;[A] is the 3x3 Σx matrix, and [B] is used for boththeΣy matrix and the coefficient matrix. The last section of the program allows you to compute a y-predicted value using the coefficients.

EXAMPLE:-

Fit a second-order curve to the following set of data and predict y when x=2.2 and whenx=3.7

X	1	2	3	4
Y	1.2	3.9	9.3	15.8

Input	Key	OUTPUT
	XEQ "LS2"`	"Σ+ cΣ FT"
1.2	ENTER^	
1	[A]	1.0000
3.9	ENTER%	
2	[A]	2.0000
9.3	ENTER^	
3	[A]	3.0000
15.8	ENTER^	
4	[A]	4.0000
	[E]	a0 = -4.4727 E-8
	R/S	a1 = 0.1700
	R/S	a2 = 0.9500
2.2	R/S	Y = 4.9720
3.7	R/S	Y = 13.6345

Notice that a0and a1 are both quite small; this is because the data correspond closely to a y=x^2 graph. You can now: press [C] (cΣ)to clear the summation registers and prepare the summation registers for a new set of data. As with CFIT, the [J] key displays a menu at any time without disturbing the program in any way.

```
 1  LBL "LS2"
 2  SF 27
 3  LBL C
 4  .01
 5  0
 6  LBL 00
 7  STO IND Y
 8  ISG Y
 9  GTO 00
10  CLST
11  LBL J
12  "Σ+ cΣ FT"
13  PROMPT
14  LBL A
15  STO 08
16  ST+ 01
17  RCL 08
18  *
19  ST+ 02
20  RCL 08
21  *
22  ST+ 03
23  RCL 08
24  *
25  ST+ 04
26  RDN
27  ST+ 05
28  RCL 08
29  *
30  ST+ 06
31  RCL 08
32  *
33  ST+ 07
34  E
35  ST+ 00
36  RCL 00
37  RTN
38  LBL E
39  "A"
40  3.003
41  MATDIM
42  .
43  43 MSIJA
44  44 RCL 00
45  45 MSR+
46  RCL 01
47  MSR+
48  RCL 02
49  MSR+
50  RCL 01
51  MSR+
52  RCL 02
53  MSR+
54  RCL 03
55  MSR+
56  RCL 02
57  MSR+
58  RCL 03
59  MSR+
60  RCL 04
61  MS
62  62"B"
63  3
64  MATDIM
65  .
66  MSIJ
67  RCL 05
68  MSR+
69  RCL 06
70  MSR+
71  RCL 07
72  MS
73  "A,B"
74  MSYS
75  "B"
76  .
77  MSIJA
78  MRR+
79  STO 08
80  "a0="
81  ARCL X
82  PROMPT
83  MRR+
84  STO 09
85  "a1="
86  ARCL X
87  PROMPT
88  MR
89  STO 10
90  "a2="
91  ARCL X
92  PROMPT
93  LBL 01
94  RCL X
95  RCL 10
96  *
97  RCL 09
98  +
99  *
100 RCL 08
101 +
102 "Y="
103 ARCL X
104 PROMPT
105 GTO 01
106 END
```

Over-conditioned Simultaneous Equations

David Hodges (DataFile V7N1p21)

System requirements: HP-41 (any model) and Advantage Module.

THEORY.

Anyone who solves simultaneous equations regularly will have encountered a situation mathematicians call over-conditioning, which means that there are more equations than there are unknowns. Suppose for example you are given the following set of equationsand areasked to find the values of x and. y:

2x + 3y = 8
3x + 4y = 10.9
4x + 5y = 14.1
5x + 6y = 16.9

At first sight this problem looks trivial. Since only two equations are required to solve tor two unknowns, it's tempting to take the two simplest equations and use then to find values for x and y. Unfortunately, that is an oversimplification because the results depend on the equations chosen. Solving the top two equations, for example, gives x = 0.7 and y = 2.2. Taking the middle two gives x = -0.1and y = 2.9; whilst the bottom two give x = 1.9 and y = 1.3.

Closer inspection of the equation system shows that none of these results are correct. If the second equation equaled 11, the third 14 and the fourth 17, then x would equal 1 and y would equal 2 no matter which pair of equations was chosen. The true values of x and y must therefore be close to 1 and 2 respectively.

The best way to deal with such a situation is to use the method of least-squares. This technique is actually equivalent to n-dimensional linear regression, where n is the number of unknowns, in this case two. Re-writing the system in matrix form, [A] * [X] = [B] {i}

$$\begin{bmatrix} 2 & 3 \\ 3 & 4 \\ 4 & 5 \\ 5 & 6 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 8 \\ 10.9 \\ 14.1 \\ 16.9 \end{bmatrix}$$

Solving this matrix equation requires two steps. The first involves pre-multiplying both sides of {i} by [A'], the transpose of matrix[A]. Next the equation is inverted to give an expression for [X]in terms of [A] and [B].[X] is then found by pre-multiplying [B] by a single matrix computed, from [A'].

Pre-multiplying{i} by [A'], and inverting for [X]:

[A'].[A].[X] =[A'].[B] {ii}
[X] = Inv**(** [A'].[A]**)**[A'].[B] {iii}

Where the notation Inv() denotes the inverse of the matrix inside the parentheses.

Inv**(**[A'].[A]**).**[A'] in {ii} may be expressed as a single matrix [C] which gives [X]directly from [B].

[X] = [C].[B] {iv}

[C] has the same number of columnsas [A] has rows, and the same number of rows as [A] hascolumns.

PROGRAMS.

Two programs are provided for the HP-41 and Advantage Module. The first, **ML** (Matrix Least-squares), computes [C] given [A]; and the second, **OS** (Over-conditioned Simultaneous equations), computes [X] [B] and the stored value of [C].

Both routines use main memory data registers to store the matrix elements. If [A] has r rows and c columns, a total of c(2r + c) + 5 data registers is required. Owners of an Extended Functions/Memory Module or and HP-41CX could write alternate versions of ML and OS which use extended memory to store the matrices. These programs could be somehow shorter, since a large proportion of the code given here isused to determine the number of registers required and to construct the matrix names.

EXAMPLE. Input dimensions of [A]rrr.ccc where r = rows and c =columns) and run ML. Input elements of matrix [A]. Perform the following steps:

INPUT	KEYS	OUTPUT
4.002	XEQ "ML"	C 1 = 0.000000
2	R/S	C2 = 0.000000
3	R/S	2: 1 = 0.000000
3	R/S	2:2 = 0.000000
4	R/S	3:2 = 0.000007
5	R/S	4: 1 = 0.000000
5	R/S	4:2 = 0.000007

View elements of matrix [C].

6	R/S	C 1 = -1.600007
	R/S	C2 = -0.700007
	R/S	C3 = 0.200007
	R/S	C4 = 1.100007
	R/S	2: 1 = 1.300007
	R/S	2:2 = 0.600007
	R/S	2:3 = -0.100007
	R/S	2:4 = -0.800007

Matrix [C] is therefore:

$$\begin{bmatrix} -1.6 & -0.7 & 0.2 & 1.1 \\ 1.3 & 0.6 & -0.1 & -0.8 \end{bmatrix}$$

Run program OS and input elements of matrix [B]**x = 0.98 ;y= 2.01**

	XEQ "OS"	C 1 = 0.000007
8	R/S	C2 = 0.000007
10.9	R/S	C3 = 0.000007
14.1	R/S	C4 = 0.000007
16.9	R/S	C 1 = 0.980007
	R/S	C2 = 2.010007

In the above examp1e, it is assumed that all allocated data registers contain zero before ML is executed. If they do not, their contents will be displayed in the prompts and will be overwritten by the data entered from the keyboard. Registers 00 and 01 are used to store the matrix names (R followed by one or more digits) and flag 00 contro1s the status of OS.

Program Listing.

```
01  LBL "OCS+"
02  "#EQS=?"
03  PROMPT
04  "#VARS=?"
05  PROMPT
06  E3
07  /
08  +
09  LBL "ML"
10  CF 00
11  "R2"
12  MATDIM
13  .9
14  1/X
15  RND
16  -9
17  *
18  E1
19  +
20  LOG
21  CHS
22  X<>Y
23  INT
24  LASTX
25  FRC
26  STO 01
27  E3
28  *
29  ST+ 01
30  *
31  RCL X
32  3
33  +
34  DIM?
35  FIX 0
36  CF 29
37  "R"
38  ARCL Y
39  MATDIM
40  ASTO 00
41  RDN
42  +
43  E
44  +
45  "R"
46  ARCL X
47  RCL 01
48  MATDIM
49  ASTO 01
50  FIX IND Z
51  SF 29
52  CLA
53  "R2"
54  XROM "MEDIT"
55  CLST
56  "`,"
57  ARCL 00
58  MMOVE
59  "R2"
60  TRNPS
61  "`,"
62  ARCL 01
63  M*M
64  CLA
65  ARCL 01
66  "`,R2"
67  MSYS
68  "R2"
69  LBL 05
70  XROM "MEDIT"
71  GTO 05
72  LBL "OCS"
73  FS? 00
74  GTO 00
75  "R2"
76  DIM?
77  ENTER^
78  FRC
79  ST- Y
80  E3
81  *
82  STO Z
83  *
84  4
85  +
86  STO Z
87  +
88  E3
89  /
90  +
91  0
92  LBL 01
93  STO IND Y
94  ISG Y
95  GTO 01
96  CLA
97  ARCL 00
98  DIM?
99  INT
100 MATDIM
101 CLA
102 ARCL 01
103 DIM?
104 INT
105 MATDIM
106 SF 00
107 LBL 00
108 CLA
109 ARCL 00
110 XROM "MEDIT"
111 "R2,"
112 ARCL 00
113 "`,"
114 ARCL 01
115 LBL 02
116 XROM "MEDIT"
117 GTO 02
118 END
```

Jacoby Method for Symmetric Matrices.

Valentín Albillo, PPCTN V1N3

For symmetric matrices the Jacobi algorithm provides a faster method. **JACOBI** was written by Valentín Albillo, and published in PPCTN, V1N3 (October 1980). I've only slightly adapted it to the SandMatrix, but basically remains the same as originally written. The paragraphs below are directly taken from the above reference to explain its workings.

This program computes all eigenvalues of a real symmetric matrix up to 22 x 22. It uses the Jacobi method, which annihilates in turn selected off-diagonal elements of the given matrix **A** using elementary orthogonal transformations in an iterative fashion, until all off-diagonal elements are zero when rounded to a given number of decimal places. Then the diagonal values are the eigenvalues of the final matrix.

The method explained. The Jacobi method does not attempt to solve the characteristic equation for its roots. It is based in the fact that a n x n symmetric matrix has exactly n real eigenvalues. Given **A**, another matrix **S** can be found so that: $\mathbf{S\,A\,S^T = D}$ is a diagonal matrix, whose elements are the eigenvalues of **A**.

The Jacobi method starts from the original matrix **A** and keeps on annihilating selected off-diagonal elements, performing elementary rotations. Let's single out an off-diagonal element, say a_{pq},, and annihilate it using an elementary rotation. The transformation R is defined as follows:

Rpp = cos z ; Rpq = sin z ; Rqp = -sin z ; Rqq = cos z
Rii = 1 ; Rpk = Riq = Rik = 0 ; for i#p,q and k#p,q

Let's now denote: $\mathbf{B = R^T\,A\,R}$, which elements are as follows:

$b_{ip} = a_{ip} \cos z - a_{iq} \sin z$

$b_{iq} = a_{ip} \sin z + a_{iq} \cos z$

$b_{ik} = a_{ik}$; where i,k # p,q

$b_{pp} = a_{pp} \cos^2 z + a_{qq} \sin^2 z - 2\, a_{pq} \sin z \cos z$

$b_{qq} = a_{pp} \sin^2 z + a_{qq} \cos^2 z + 2\, a_{pq} \sin z \cos z$

$b_{pq} = 0$, and the remaining elements are symmetric.

where: sin z = w / sqrt(2(1+sqrt(1-w^2))), and cos z = sqrt (1-sin^2 z)

with: L = - a_{pq}, M = (a_{pp}-a_{qq}) / 2 , and w = L sign(M) / sqrt (M^2+L^2)

This is iterated using a strategy for selecting each non-diagonal element in turn, until all non-diagonal elements are zero when rounded to a specific number of decimal places. When this is so, the diagonal contains the eigenvalues.

Program remarks. The accuracy and running times are display settings-dependent, however the computed eigenvalues are very often more accurate that it'd appear; for instance, using FIX 5 it's quite possible to have eigenvalues accurate to 8 decimal digits. The program is written to be as fast as possible and to occupy the minimum amount of program memory; the matrix is stored taking into account its symmetry, so that all elements are stored only once (as aji = aij). For a nxn matrix minimum size is [½ (n^2 + n) + 7].

Example. Find the eigenvalues for the 4x4 matrix: A =

```
[[ 25   -41    10    -6]
 [-41    68   -17    10]
 [ 10   -17     5    -3 ]
 [ -6    10    -3     2 ]]
```

Keystrokes	Display	Result
XEQ "**JACOBI**"	ORDER=?	Prompts for dimension
4, R/S	a 1,1=?	Data entry starts
25, R/S	a 1,2=?	
41, CHS, R/S	a 1,3=?	
10, R/S	a 1,4=?	
6, CHS, R/S	a 2,2=?	Note how the symmetric
68, R/S	a 2,3=?	elements are skipped
17, CHS, R/S	a 2,4=?	
10, R/S	a 3,3=?	
5, R/S	a 3,4=?	
3, CHS, R/S	a 4,4=?	input the last element
2, R/S	PREC.=?	Asks for precision
5, R/S	RUNNING..	Scrolling on the display
	x=0,033027	
R/S	x=98,52 170	After a while ~ 2.5m in normal 41
R/S	x=1,18609	the four ev's are displayed.
R/S	x=0,25920	

The characteristic polynomial can be found using CHRPOL in the SandMatrix, resulting:

Chr(A) = x^4 -100 x^3 + 146 x^2 – 35 x +1

Program Listing.

```
01  LBL "JACOBI"
02  RAD
03  03 "ORDER=?"
04  PROMPT
05  ENTER^
06  ENTER^
07  X^2
08  +
09  2
10  /
11  7
12  +
13  SIZE?
14  X<>Y
15  X>Y?
16  PSIZE
17  RCL Z
18  STO 00
19  E3/E+
20  STO 02
21  7
22  STO 04
23  23*LBL 12
24  RCL 02
25  STO 03
26  *LBL 07
27  "a"
28  RCL 02
29  AINT
30  "`:"
31  RCL 03
32  AINT
33  "`=?"
34  PROMPT
35  STO IND 04
36  ISG 04
37  X<>Y
38  ISG 03
39  GTO 07
40  ISG 02
41  GTO 12
42  "PREC.=?"
43  PROMPT
44  FIX IND X
45  -NL SYSTEMS
46  *LBL 70
47  2
48  STO 03
49  *LBL 85
50  E
51  *LBL 87
52  STO 02
53  CF 00
54  RCL 03
55    XEQ 90
56    X<>Y
57    RND
58    X=0?
59    GTO 84
60    SF 00
61    LASTX
62    ST- IND Z
63    STO 01
64    ST+ 01
65    CHS
66    STO 06
```

```
67   RCL 02
68   RCL 02
69   XEQ 90
70   STO N
71   RDN
72   RCL 03
73   RCL 03
74   XEQ 90
75   STO O
76   RDN
77   -
78   STO M
79   2
80   /
81   SIGN
82   RCL 06
83   LASTX
84   X^2
85   RCL 06
86   X^2
87   +
88   SQRT/
89   *
90   E
91   RCL Y
92   X^2
93   -
94   SQRT
95   E
96     +
97   ST+ X
98   SQRT
99   /
100  STO 06
101  ST* 01
102  X^2
103  ST* M
104  E
105  STO 04
106  X<>Y
107  -
108  SQRT
109  ST* 01
110  X<> M
111  RCL 01
112  +
113  ST- IND N
114  ST+ IND O
115  *LBL 01
116  RCL 03
117   RCL 04
118   X=Y?
119   GTO 08
120   RCL 02
121   X=Y?
122   GTO 08
123   XEQ 90
124   STO 05
125   RDN
126   STO N
127   RCLM
128   STO O
129   *
130   RCL 03
131   RCL 04
132   XEQ 90
133   STO T
134   RDN
135   ST* O
136   RCL 06
137   ST* N
138   *
139   -
140   STO IND 05
141   RCL N
142   RCL O
143   +
144   STO IND Z
145   RCL 04
146   RCL 00
147   X<=Y?
148    GTO 84
149    *LBL 08
150    E
151    ST+ 04
152    RCL 00
153    RCL 04
154    X<=Y?
155    GTO 01
156    *LBL 84
157    RCL 03
158    RCL 02
159    E
160    +
161  X#Y?
162  GTO 87
163  X<>Y
164  RCL 00
165  X#Y?
166  ISG 03
167  X<>Y
168  X#Y?
169  GTO 85
170  FS?C 00
171  GTO 70
172  RCL 00
173  E3/E+
174  STO 06
175  TONE 3
176  *LBL 13
177  "X="
178  RCL 06
179  INT
180  ENTER^
181  XEQ 90
182  X<>Y
183  ARCL X
184  PROMPT
185  ISG 06
186  GTO 13
187  RTN
188  *LBL 90
189  X>Y?
190  X<>Y
191  RCL 00
192  ST+ X
193  X<>Y
194  -
195  E
196  ST- L
197  X<> L
198  *
199  2
200  /
201  +
202  6
203  +
204  RCL IND X
205  X<>Y
206  END
```

2. SandMatrix Applications

Anti-Identity Matrix {AIM}

Here's a song to the unsung-hero: meet the anti-identity matrix, a negative version (its evil twin perhaps?) of the much-better known "Identity" matrix – with all elements reversed, i.e. zeroes instead of ones and vice-versa.

Why bother, you would ask? Well, having a routine to create anti-IDN (a.k.a. AID) matrices comes very handy to test many of the routines included in the SandMatrix and in this very module as well, so here it is for your utter enjoyment.

These matrices can be easily constructed using the SandMatrixfunction **MZDG** , which only deletes the diagonal elements,whenapplied to a all-ones matrix - so using the three-step sequence:

{ 1, **MCON**, **MZDG** }.

These matrices have the interesting (unproven) property that their determinants obey the expression:

Det [AID(nxn)] = (-1)^(n-1) . (n-1)

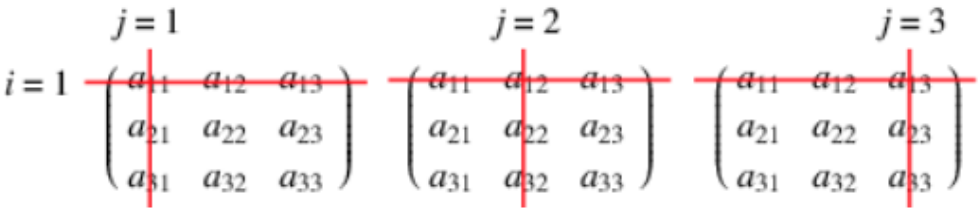
To test it, creating a 30x30 AIM can't be simpler (if you use the CL Y-Memory area, that is): just create the matrix first with **MATDIM** (needs name in ALPHA), input the order in X and call the routine. You should have something like this:

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
2	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
3	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
4	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
5	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
6	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
7	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
8	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
9	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
10	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
11	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
12	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
13	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
14	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
15	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
16	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1
17	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1
18	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1
19	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1
20	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1
21	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1
22	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1
23	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1
24	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1
25	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1
26	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1
27	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1
28	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1
29	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1
30	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0

Matrix Minors& Sub-matrices. {MINOR , MINORS , SUBMAT }

In linear algebra, a minor of a matrix [A] is the determinant of some smaller square matrix, cut down from [A] by removing one or more of its rows or columns. Minors obtained by removing just one row and one column from square matrices (first minors) are required for calculating matrix cofactors, which in turn are useful for computing both the determinant and inverse of square matrices.

If [A] is a square matrix, then the minor of the entry in the i-th row and j-th column (also called the (i,j) minor, or a first minor[1]) is the determinant of the sub-matrix formed by deleting the i-th row and j-th column. This number is often denoted Mi,j. The (i,j) cofactor is obtained by multiplying the minor by: (-1)^{i+j}.



Two programs are included, one for Real matrices (not limited in order, courtesy of **MDET**) and another for Complex Matrices – only up to degree 5, due to the restriction imposed by **CMDET**. The programs are a good example of utilization of the utility functions **C<>C**, **R<>R**, and **MMOVE.**

Example: Calculate all element minors for the example matrix below:

```
1  2  3
4 -5  6
7  8  9
```

You need to provide the matrix name in ALPHA *pointer value in X* -i.e. from 1,001 to 3,003 in this example. You can do it one at a time using **MINOR**, or all sequentially using **MINORS.** The latter option will create a new matrix in X-Mem named "MINORS", with all elements being the minors of the original matrix.

The solutions are:

```
          -3   -6   -3
Minors:   -6  -12   -6
          -3   -6   -3
```

You can also use **SUBMAT** to reduce the matrix one unit on each dimension, based on the element of your choice removing its row and column. For instance using i,j = 2,001 in X and executing SUBMAT will change the original matrix into a 2x2 submatrix, with the following elements:

```
2  3
8  9
```

Warning: **SUBMAT replaces the original matrix with the reduced one.** If you need the original matrix to remain in memory *you must copy it with a different name before calling* **SUBMAT***to safeguard it.*

Program listing.- Real Matrix Minors

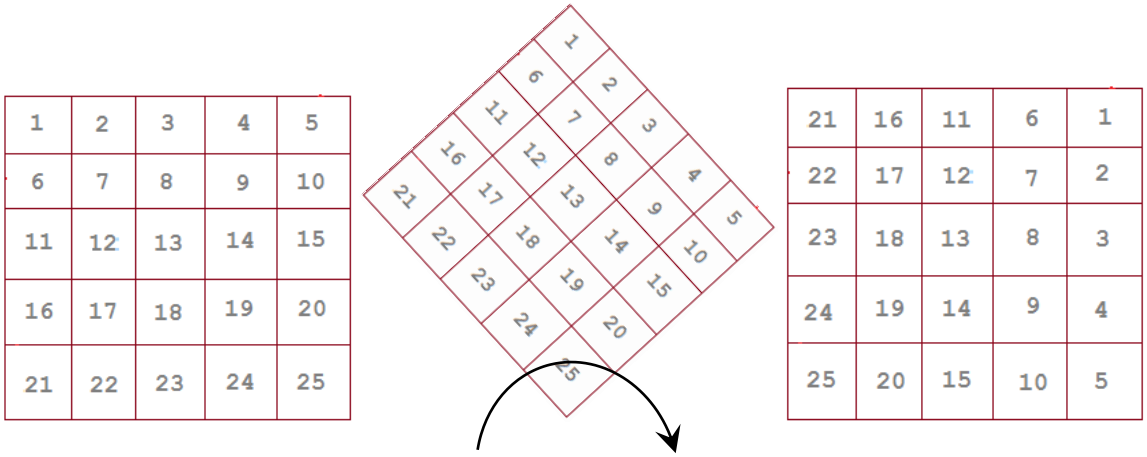
1	LBL "MINOR"		28	MNAME?	
2	LBL 01		29	RTN	
3	ASTO 01	*MNAME*	30	*GTO 01*	
4	STO 00	*i,j pointer*	31	LBL 02	
5	*"|-,#1"*		32	INT	*j*
6	MAT=	*scratch copy*	33	ENTER^	
7	DIM?		34	DSE X	*j-1*
8	1,001		35	X=0?	
9	-	*one order less*	36	RTN	*don't bother if j=1*
10	*"#2"*		37	X<>Y	
11	MATDIM	*scratch sub-array*	38	ENTER^	
12	MZERO	*clear it*	39	ENTER^	
13	*"#1"*		40	I<>J	*0,00(j-1)*
14	RCL 00		41	E	
15	I<>J	*i,j pointer*	42	-	
16	SF 00		43	+	*j,00(j-1)*
17	XEQ 02		44	LBL 00	
18	RCL 00		45	FS? 00	
19	CF 00		46	C<>C	*bubble left column*
20	XEQ 02		47	FC? 00	
21	CLST		48	R<>R	*bubble up row*
22	2,002		49	1.001	*offset*
23	*"#1,#2"*		50	-	*k,00(k-1)*
24	MMOVE		51	DSE Y	*j=j-1*
25	PURFL		52	*GTO 00*	
26	CLA		53	**END**	
27	MDET				

Matrix Rotation &Mirror Images. {M90R, M90L, MMIRR}

A 90-degree clockwise rotation pivots the complete matrix around its bottom-right element, i.e the last element in the last column works as the rotation "axis" - whilst a counter-clockwise 90-deg turn uses the bottom-left element, i.e. the last element in the first column

This type of rotations is the simplest one to implement, thanks to the row or column swapping functions (depending of the direction of the rotation), applied on the transposed matrix. The algorithm consists of successive row or column switches done on the transposed matrix, and thus it's faster than using an individual element mapping for each of the layers (or "rings") in the matrix – which is also dependent on the matrix dimensions.

For example, rotating the 4x4 matrix below 90 degrees *clockwise*; see how the rotated matrix is the vertical-mirror image of the original transposed?

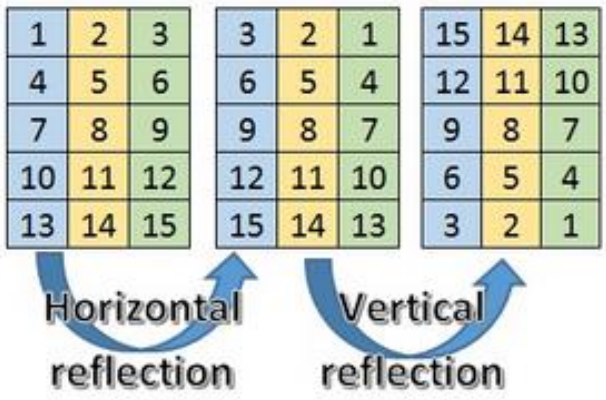


Similarly, a counter-clockwise 90-deg rotation is the horizontal-mirror image of the original transposed.

Our routines will simply transpose the matrix first, and then call the mirror image routine – consisting of a row or column swapping repeated as many times as columns are in the transposed matrix.

Terminology alert:

"Reflection" is the analogous term to mirror image, although the horizontal and vertical reflections can be confusing since they use vertical and horizontal "mirrors", which is intuitively the opposite.

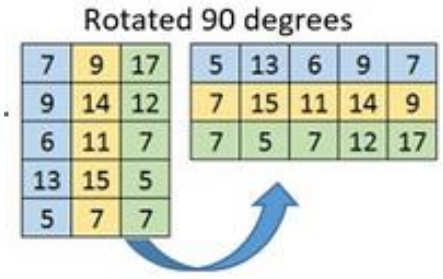


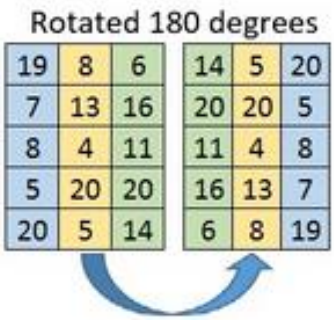
Program usage: Just type the matrix name in ALPHA and execute **M90R** or **M90L** depending on the desired direction of rotation. For **MMIRR** you need to clear or set user flag manually to indicate vertical or horizontal mirror image respectively.

Program Listing:

```
01  LBL "M90R"     ; right
02  CF 00          ; flag case
03  GTO 00         ; merge
04  LBL "M90L"     ; left
05  SF 00          ; flag case
06  *LBL 00        ; common
07  TRNPS          ; transpose
08  LBL "MMIRR"    ; mirror
09  DIM?           ;n x m
10  FRC            ; 0,00m
11  E
12  +              ; 1,00m
13  *LBL 01
14  FC? 00         ; right?
15  C<>C
16  FS? 00         ; left?
17  R<>R"
18  E-3            ; next col?
19  -
20  ISG X          ; next row
21  GTO 01         ; repeat
22  END            ; done
```

Note: Refer to the 'Complex Matrix" ROM for additional functions based on a single-element rotation, also for Real matrices but unfortunately there was no room available in this module to include them here.





Matrix L/U Editing. {MSYM, MU<>L}

Two small routines to edit the L/U sections of a square matrix, as follows:

- **MSYM** copies the Upper region into the lower one, so the matrix becomes symmetric.
- **MU<>L** exchanges the upper and lower regions.

These routines don't have a prominent applicability but are a good example of utilization of the element manipulation functions in the SandMatrix, specifically **MXIJ**– the in-place pointer index exchange.

MXIJ facilitates the element transposition by exchanging the row and column of the currently selected element, returning the new selected element pointer to the X-Register. The matrix can be non-square, but an error message will show if the "transposed" pointer does not exist. Note that there's no need to recall the current pointer first.

The function does the equivalent to the following FOCAL snippet: { MRIJ, I<>J, MSIJ }, which is simple enough but having it as a single function allows simplified FOCAL programs and doesn't disturb the stack.

Program listing:

```
 1  LBL "MSYM"
 2  CF 00
 3  GTO 00
 4  LBL "MU<>L"
 5  SF 00
 6  *LBL 00
 7  ,
 8  MSIJA
 9  *LBL 01
10  MRIJA
11  INT
12  I<>J
13  LASTX
14  +
15  MSIJ
16  J+
17  FC? 10
18  RTN
19  *LBL 02
20  MR
21  MXIJ
22  FC? 00
23  XEQ 03
24  FS? 00
25  XEQ 04
26  J+
27  FC? 09
28  GTO 02
29  GTO 01
30  *LBL 03
31  X<>Y
32  MS
33  MXIJ
34  RTN
35  *LBL 04
36  RDN
37  MR
38  XEQ 03
39  X<> Z
40  MS
41  END
```

Row Division by Diagonal element. (Diagonal Unitary) { R/aRR }

This function is used to modify the values of all elements, dividing each row by its diagonal element; that is: aij = aij / aii, j=1,2,... n

In effect the result matrix has all its diagonal elements equal to 1 (i.e. diagonal is unitary). This type of calculation is useful for row simplification steps in matrix reductions; more like a vestigial function from when the major matrix operations were not available (i.e. the CCD days, pre-Advantage Pac).

Program listing:

1	LBL "R/aRR"	*MNAME in Alpha*	19	RDN	*discard product*
2	SQR?	*square?*	20	FC? 09	*end of row?*
3	LU?	*yes but LU?*	21	*GTO 00*	*no, get next element*
4	-ADV MATRX	*not square, show error*	22	FS? 10	*end of matrix?*
5	0		23	*GTO 02*	*yes, exit*
6	MSIJA	*set pointer to 1:1*	24	MRIJ	*recall pointer*
7	LBL 01		25	ENTER^	
8	MR	*recall diag element*	26	INT	
9	1/X	*inverse value*	27	ENTER^	
10	X<>Y	*pointer to X*	28	I<>J	*does E3/ if integer*
11	MSIJ	*set pointer*	29	+	*j,00j*
12	X<>Y	*value back to X-reg*	30	MSIJ	*set pointer*
13	ENTER^		31	X<>Y	
14	ENTER^	*fill stack w/ value*	32	*GTO 01*	*next row*
15	LBL 00		33	LBL 02	
16	MR	*recall element*	34	DIM?	*get dimansion*
17	*	*multiply*	35	END	*end*
18	MSR+	*store and increase column*			

Note:

This routine was moved to this ROM to make room in the SandMatrix for CMTRC, the Complex Matrix Trace routine. This arrangement is more self-contained, favouring the SandMatrix capability to support the Complex Determinant by itself (CMTRC is used as a subroutine by CMDET).

Appendix. Harmonic Determinants. {DNN , DN }

This section reflects the discussion started by Valentín Albillo on the HP-Museum forum. It's useful to showcase the capabilities of the CL_Y-Registers for very large size matrices.

Consider the determinant D(N) defined as follows:

```
3    1    1 ... 1
1    4    1 ... 1
1    1    5 ... 1
... ... ... ... ...
1    1    1 ... N+2
```

This type of determinants have an exact formula using the Harmonic function, H(N):

D(N) = (N+1)! . H(N)

The sum of harmonic series is thus: H(N) = D(N-1) / N!
which surely would be one of the most inefficient ways to compute it ;-)

Using the CL_Y-Registers area, write a routine to compute D(N) – and verify the direct formula for the values N=11, 13, 30, 40 and N=55 - which will use 3,025 Y-Registers.

The routines are listed below. Both expect the order N in the X-register:

```
01 LBL "DDN"
02 RCL X
03 E3
04 /
05 +
06 "Y"- matrix will start at RY-001
07 MATDIM
08 E
09 MCON
10 CLX
11 MSIJA
12 2
13 LBL 00
14 E
15 +
16 MSC+
17 SF 25
18 J+
19 FS?C 25
20 GTO 00
21 MDET
22 END
23 LBL "DN"
24 E
25 +
26 HARM
27 LASTX
28 FACT
29 *
30 END
```

And the table below shows the results from each approach:

N	D(N)	Time (@Turbo50)	Formula
11	1,486,442,880.0	1.8 sec	1,486,442,880.0
13	2.834656472 E11	2.01 sec	2.834656474 E11
30	3.311538747 E34	11 sec	3.311538746 E34
40	1.439439902 E50	1 min 20 sec	1.439439902 E50
50	3.278748200 E75	2 min 30 sec	3.278748199 E75

Warning: *Remember that the CL is required to store a matrix in the Y-Registers area*. Otherwise you'll get the error message on the right:

3. Recursive 2D-Solve & Integrate

Recursive Utilization of FINTG and FROOT.

Like the original SOLVE and INTEG did, both **FROOT**& **FINTG** support "crossed" nested calls from one another, i.e. you can call FROOT from an integrand function being used by FINTG, and you can call FINTG in the root-finding function definition for FROOT. However, it is not possible to recursively call either one of these functions sequentially from within a FOCAL routine. Any attempts to do so triggers the "*RECURSION*" error message and the execution aborts.

The SIROM provides a set of MCODE functions and two FOCAL routines to overcome this limitation. Each time FROOT/FINTG is executed it creates a dedicated memory buffer to store the application data and to perform all the math. The basis of the operation is the use of a secondary memory area for the nested call of the function, not conflicting with the initial memory buffer created in the first call. The main loop uses the initial buffer #14, and the operand function in turn creates a secondary buffer #14 to use for the nested loop – deleting it after it's complete.

In order to reuse the existing code, we'll trick the OS changing the id# of the initial buffer #14 right before the second call – *not deleting it but cloaking it in the I/O Memory area of the calculator*. The operand function re-labels the buffer with id#13 (using function **CLOAK**), then the nested call to FROOT/INTEG creates and uses a new buffer #14 to perform its task and deletes it upon completion – returning the execution to the "operand" function FOCAL routine. Before the execution is returned to the driver program, the cloaked buffer is re-issued as id#14 (using function **EXPOSE**) so things can be picked up exactly where there were left off before calling the nested subroutine.

If you must know, all **CLOAK** and **EXPOSE** do is changing the buffer id#' of the initial buffer created in the first call to FROOT/INTEG - first from 14 to 13, and then back to 14. Prior to all this a third function (**RESET**) is used to check for pre-existing buffers with id#13 – deleting it if found.

2D Driver Routines and Rules of Engagement.

The main programs for double integrals and system of 2 equations are **FITG2** and **FRT2**. Each one has an auxiliary routine associated with it, which acts as the first level operand function and issues a second nested call for the integrand or the second equation appropriately, as follows:

For **FITG2**, the function name f(x,y) is expected in ALPHA, and the four integral limits in the stack in the pattern "y1, y2, x1, x2" – (y1,y2) for the outer integral, and (x1,x2) for the inner one.

- *The integrand function is to be programmed assuming x is in R01, and y in the stack.*

For **FRT2**, both function names are expected to be in Alpha separated by comma (like "F1,F2"), and the guesses entered in the stack, with the pattern "x1, x2, y1, y2" - with (x1, x2) for f1(x,y) and (y1, y2) for f(2(x,y).

- *The second operand function f2(x,y) is executed first. It assumes x in R01 and y in the stack.*
- *The first operand function f1(x,y) assumes x in R01 and y in R02.*
- *You decide which one is F1 and F2 by their order in the ALPHA string*

All buffer management is made automatically by the auxiliary routines ***2D** and***FG**.

Routine Listings.

Here are the routine listings for your perusal. Notably **FRT2** introduces more complexity to process the function names – entered as comma-separated strings in ALPHA – and due to the indirect call to f1(x,y) at the end of the auxiliary routine ***FG** - which is not required by ***2D** in the double integration case, as it's just one function involved. **CLAC** and **ASWAP** are borrowed from the ALPHA ROM – and need the Library#4 present in the calculator. They're only used for **FRT2**.

01	**LBL "FRT2"**	
02	**CLKEYS**	*no keys assigned*
03	**ASTO 00**	*save string*
04	**ASWAP**	*swap around ","*
05	**CLAC**	*remove second*
06	**ASTO 05**	*save in R05*
07	**CLA**	
08	**ARCL 00**	*recall string*
09	**CLAC**	*remove second*
10	**ASTO 00**	*save in R00*
11	**STO 04**	*upper guess2*
12	**RDN**	
13	**STO 03**	*lower guess2*
14	**RDN**	
15	**RESET**	*reset buffers*
16	***"*FG"***	*first level operand*
17	**FROOT**	*call first round*
18	**GTO 00**	
19	***LBL 01**	*Not found*
20	**RESET**	
21	*"NO ROOT"*	
22	**AVIEW**	
23	***LBL 00**	*Found*
24	**RCL 02**	*y solution*
25	**X<>Y**	*arrange in stack*
26	**CLA**	*appends*
27	**ARCL 00**	*f1(x,y) name*
28	*"|-,"*	
29	**ARCL 05**	
30	**RTN**	*done(!)*
31	***LBL "*FG"**	
32	**STO 01**	*save x for later*
33	**CLOAK**	*mask buffer id#*
34	**RCL 03**	*lower guess 2*
35	**RCL 04**	*upper guess 2*
35	**CLA**	
36	**ARCL 05**	*f2(x,y)*
37	**FROOT**	*nested call*
38	***GTO 00***	*Found yo, skip*
39	***GTO 01***	*Not found!*
40	***LBL 00**	
41	**EXPOSE**	*re-issue buf id#*
42	**STO 02**	*Save yo result*
43	**XEQ IND 00**	*calculates f1(x,Yo)*
44	**END**	

01	***LBL " FITG2"**	
02	**CLKEYS**	*no keys assigned*
03	**ASTO 00**	*save in R00*
04	**STO 03**	*upper limit2*
05	**RDN**	
06	**STO 02**	*lower limit2*
07	**RDN**	
08	**RESET**	*reset buffers*
09	*"2D"*	*first level operand*
10	**FINTG**	*call first round*
11	**RTN**	done
12	*"NO SOL"*	
13	**AVIEW**	
14	**RESET**	
15	**RTN**	done.
16	***LBL "*2D"**	
17	**STO 01**	*Save x for later*
18	**CLOAK**	*mask buffer id#*
19	**RCL 02**	*lower limit2*
20	**RCL 03**	*upper limit2*
21	**CLA**	
22	**ARCL 00**	*f(x,y)*
23	**FINTG**	*nested call*
24	**EXPOSE**	*re-issue buf id#*
25	**END**	*ready*

FITG2 uses registers {R00-R03} and leaves the results in X and R01. The function name is left in ALPHA (6-chars max).

FRT2 uses registers {R00-R05} and leaves the results in the stack registers {X, Y} and {R01, R02} for the 2-equation roots. The comma-separated function names string is left in ALPHA (6-chars max for each name).

Comments.

The new functions to support the nested configuration are simplified versions of some general-purpose buffer utilities, available in other extension modules as follows:

- **RESET** is equivalent to the sequence { 13, **B?**, **CLB,** RDN }
- **CLOAK** is equivalent to the sequence { 14.013 , **REIDBF**, RDN}
- **EXPOSE** is equivalent to the sequence: { 13.014 , **REIDBF ,** RDN }

B? and **CLB** are available in the OS/X ROM, and **REIDBF** in the RAMPAGE ROM.

Using the simplified versions is more intuitive for math-oriented users, and besides freed up some space for additional examples in the SIROM.

While you can use **RESET** at any time (which will delete buff #13 if present, or do nothing if not present), using **CLOAK** and **EXPOSE** will generally result in the error message "BUF ERR". They're meant to be used only while buffer #14 exists, which is tightly controlled by the code in FINTG and FROOT – and furthermore, the SIROM uses the I/O_PAUSE interrupt as a "search & destroy" for buffer#14 at all times. Refer to the corresponding section in the **SandMath** manual to read more on this subject.

Caveat emptor:

- There's a price to pay for this buffer trickery, and that's the loss of the USER key assignments. As you can see in the listings above, the main routines call **CLKEYS** to make the operation more reliable (this avoids spurious buffer errors due to memory overwrites). You can save them in an X-Mem file using **SAVEKA** and then recover them with **GETKA** after the fact (both functions are also available in the AMC_OS/X ROM).

- These routines are not fast, their interest is in the methodology - not optimized for speed to say the least. If you need faster responses, then the SandMath provides dedicated MCODE functions for many of these and yet some more.

- Bear in mind that the INTEG-based method to define special functions is not an efficient one from the mathematical standpoint, but it is a godsend for engineering problems. Also FROOT is not perfect or fool-proof either, so choosing a good initial guess is of high importance. If FRT2 fails to find a root (in either variable), it'll present the error message "NO ROOT" – Change the limits and try again.

The following examples should provide a good overview into the details of the programming.

Example 1. Calculate the integral of the Bessel Jn function, ITJ(1,3) = **INT** (0,3) { J(1,t).dt}
using the integral definition as reference:

$$J_n(x) = \frac{1}{\pi}\int_0^{\pi} \cos(n\tau - x\sin\tau)\,\mathrm{d}\tau.$$

Program Code is below. Note that you don't need to worry about the buffer management, that's done automatically by the driver routines all transparently to the user.

01	**LBL "ITJB"**		13	**LBL " *JN"**	*inner variable t in stack*
02	**X<>Y**	*order n to X*	14	**RAD**	*angular mode*
03	**STO 04**	*order saved in R04*	15	**RCL 04**	*get order*
04	**CLX**	*lower outer limit*	16	*	*n.t*
05	**X<>Y**	*upper outer limit*	17	**X<>Y**	*inner variable t*
06	**0**	*lower inner limit*	18	**SIN**	*sin t*
07	**PI**	*upper inner limit*	19	**RCL 01**	*outer variable*
08	*"*JN"*	*function name*	20	*	*x.sin t*
09	**XROM " ITG2"**	*double integration*	21	-	*n.t - x.sin t*
10	**PI**	*adjust factor*	22	**COS**	*cos (n.t - x.sin t)*
11	**/**	*final result*	23	**END**	*integrand complete.*
12	**RTN**	done.			

As mentioned before, *speed is not this method's forte*. Even on V41 in turbo mode it'll take a good 75 seconds to return 1.260052 (in FIX 6). This was not the goal of the example, but to clarify the general guidelines and showcase the conceptual approach. If you want a fast result you're encouraged to use **JBS** in the SandMath, or even better the **ITJ**(sub)function also in the SandMath, which uses the Generalized, Regularized Hypergeometric function for the calculation – a world of differences…

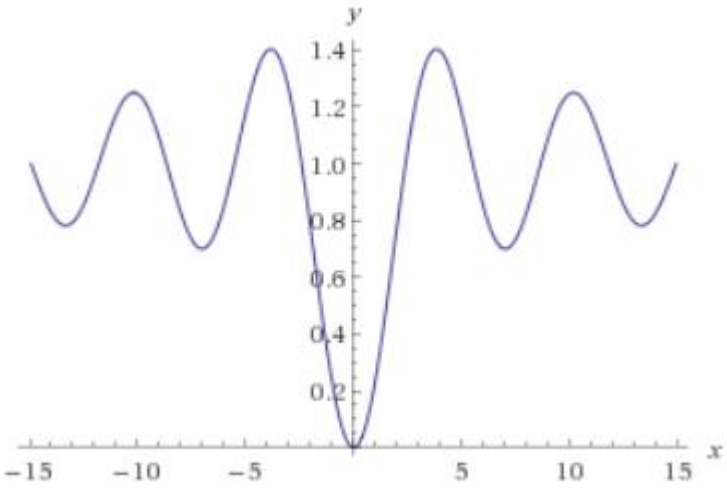
Comment. This particular example is of course much better dealt with using the well-known expression between the Bessel function J1 and J0 shown below (proving once again that it's always good to check your math before embarking in long and winding paths):

$$\int_0^x J_1(t)\,dt = 1 - J_0(x)$$

thus:

$$\int_0^3 J_1(t)\,dt = 1 - J_0(3) \approx 1.26005$$

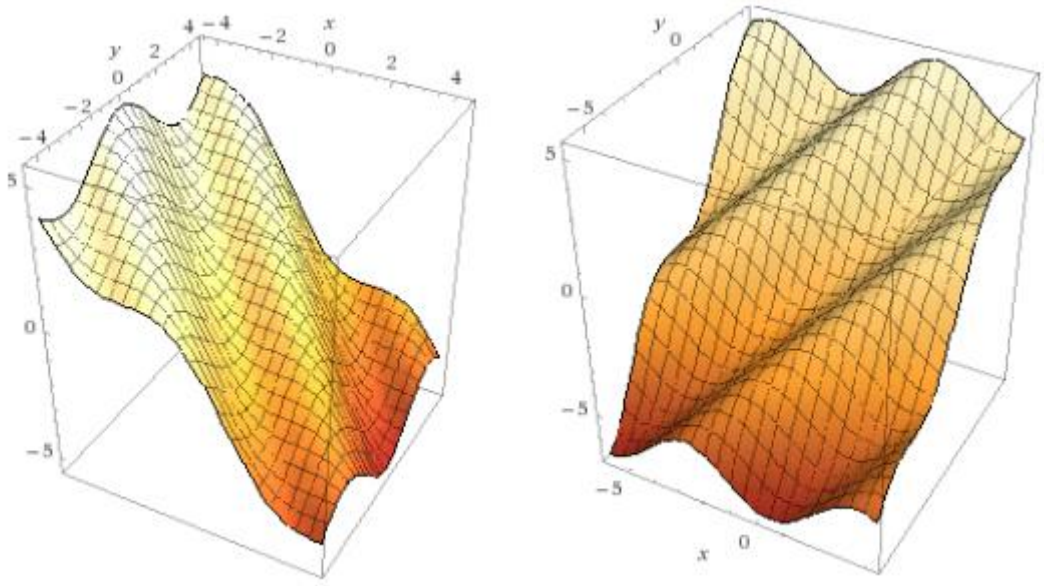
Here's an interesting plot showing the integral function of J1(x) between]-15 . 15[



Example 2. Calculate the solution for the system of non-linear equations below:

f1(x,y) =x - sin(x + y)
f2(x,y) = y - cos(x - y)

Solution: x = 0,935082064
y = 0,998020058



The equations are programmed as shown below. Note how the convention is observed, with the y value assumed in the stack for the second function and in R02 for the first one; whilst x is always assumed in R01 for both functions. The solutions are obtained in about 3 seconds (FIX 9) using V41 in Turbo mode.

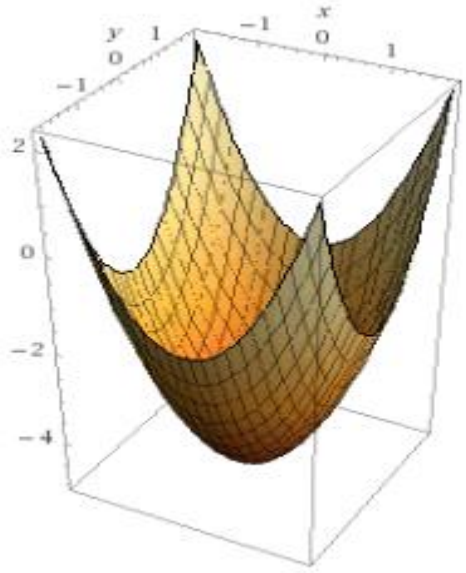
ALPHA, "F1,F2" , ALPHA, 1, ENTER^, 2, ENTER^, 1, ENTER^, 2, XEQ "FRT2"

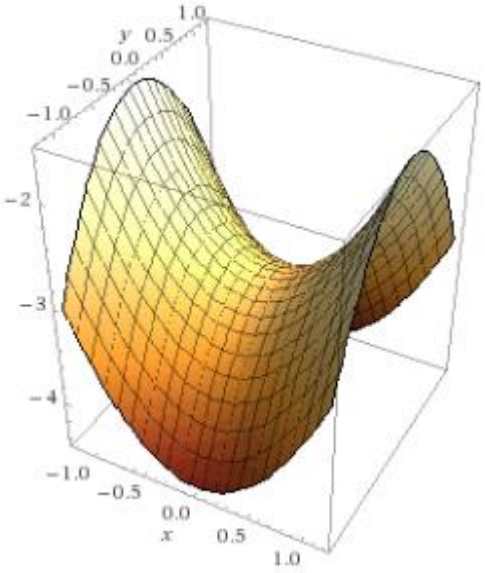
01	**LBL "F1"**	
02	**RCL 01**	*x*
03	**RCL 02**	*y*
04	**+**	*x+y*
05	**SIN**	*sin(x+y)*
06	**RCL 01**	*x*
07	**-**	*sin(x+y)-x*
08	**RTN**	

09	**LBL "F2"**	
10	**RAD**	
11	**CHS**	*-y*
12	**RCL 01**	*x*
13	**+**	*x-y*
14	**COS**	*cos(x-y)*
15	**X<>Y**	*y*
16	**-**	*cos(x-y)-y*
17	**END**	

Example 3. Obtain the roots for the system of two equations below (available as "G1" and "G2")

g1(x,y) = x^2 + y^2 -5 Solution: x = 2
g2(x,y) = x^2 -y^2 - 3 y = 1





This is an interesting case because FRT2 not only is much slower (as we knew it was going to be), but also fails to find a root using initial guesses equal to the solutions, i.e. x0 = 2, y0=1.

Other Examples.

Let's use Valentín Albillo's neat examples from DataFile for Double Integrals - as follows:

$$I = \int_0^1 \int_1^2 (x^2 + y^2)\ .dy.dx \quad ; \quad I = \int_3^4 \int_1^2 1/(x + y)^2\ .dy.dx$$

$$I = \int_{-2.3}^{1.6} \int_{3.9}^{6.1} (e^{-x*x} + x^3 - y^3 * x^2 + 7) * \tan^{-1}(x-2) * \sin(y+3)\ .dy.dx$$

See the original article for details, available at:
http://web.archive.org/web/20110906135412/http://membres.multimania.fr/albillo/calc/pdf/DatafileVA024.pdf

The results are: I1 = 8/3 = 2.6666666
I2 = Ln(25/24) = 0.040821
I3 = 1,321.275779

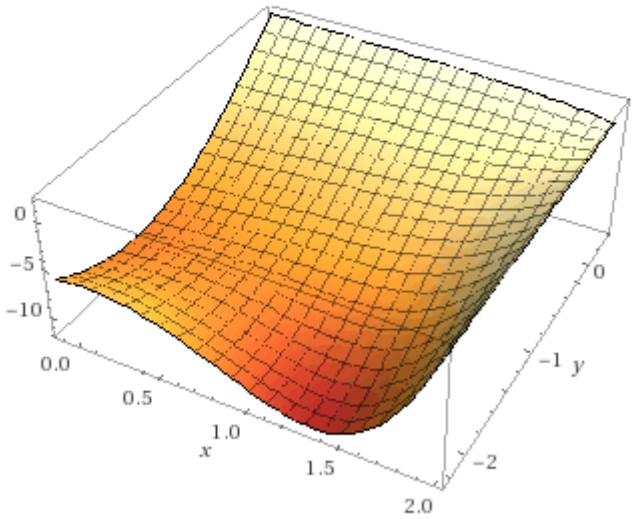
Input interpretation:

3D plot	$(\exp(-x^2)+x^3-y^3\,x^2+7)\tan^{-1}(x-2)\sin(y+3)$

$\tan^{-1}(x)$ is the inverse tangent function

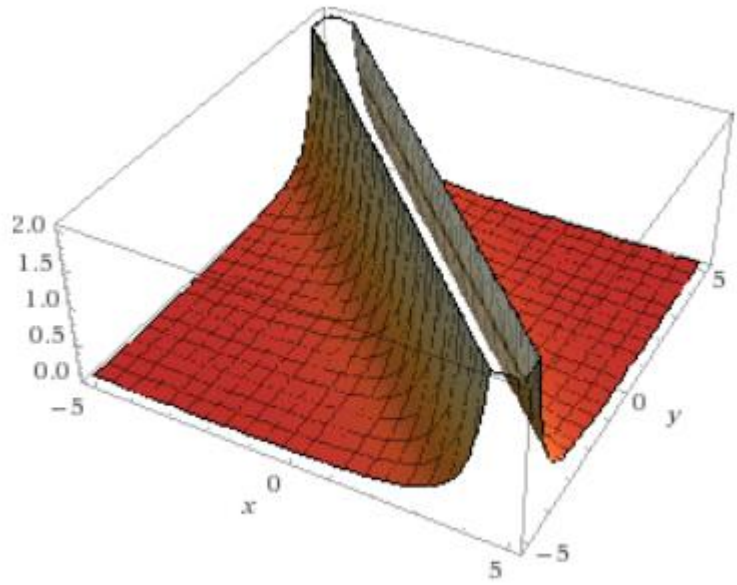
3D plot:

Show contour lines



Open code

3D plot	$\frac{1}{(x+y)^2}$



Appendix: MCODE listing for dedicated functions

BUFERR	AEC0	20D	?NC XQ	Build Msg - UF25 Clear
	AEC1	0FC	->3F83	[APERMSG]
	AEC2	002	"B"	
	AEC3	015	"U"	
	AEC4	006	"F"	"NO BUF" or
	AEC5	020	" "	"DUP BUF"
	AEC6	005	"E"	
	AEC7	012	"R"	
	AEC8	212	"R"	
	AEC9	1F1	?NC GO	LeftJ, Show and Halt
	AECA	0FE	->3F7C	[APEREX]
Header	AECB	08B	"K"	
Header	AECC	001	"A"	
Header	AECD	00F	"O"	
Header	AECE	00C	"L"	
Header	AECF	003	"C"	Ángel Martin
CLOAK	AED0	388	SETF 0	
	AED1	130	LDI S&X	
	AED2	00E	CON: 14	buffer id# = "E"
	AED3	053	JNC +10d	[MERGE]
Header	AED4	085	"E"	
Header	AED5	013	"S"	
Header	AED6	00F	"O"	
Header	AED7	010	"P"	
Header	AED8	018	"X"	
Header	AED9	005	"E"	Ángel Martin
EXPOSE	AEDA	384	CLRF 0	
	AEDB	130	LDI S&X	
	AEDC	00D	CON: 13	buffer id# = "D"
MERGE	AEDD	106	A=C S&X	
	AEDE	000	NOP	
	AEDF	2A5	?NC XQ	
	AEE0	10C	->43A9	[CHKBFA]
NOBUF2	AEE1	2FB	JNC -33d	[NOBUF]
BFOUND2	AEE2	2DC	PT= 13	
	AEE3	38C	?FSET 0	cloaking?
	AEE4	027	JC + 04	yes, skip
MAKE14	AEE5	390	LD@PT- E	change id# to "EE"
	AEE6	390	LD@PT- E	
	AEE7	01B	JNC +03	
MAKE13	AEE8	350	LD@PT- D	change id# to "DD"
	AEE9	350	LD@PT- D	
	AEEA	2F0	WRTDATA	
	AEEB	3C1	?NC GO	Normal Function Return
	AEEC	002	->00F0	[NFRPU]

Header	AEAE	094	"T"	
Header	AEAF	005	"E"	**Deletes Buffer 13**
Header	AEB0	013	"S"	
Header	AEB1	005	"E"	
Header	AEB2	012	"R"	Ángel Martin
RESET	AEB3	130	LDI S&X	
	AEB4	00D	CON: 13	buffer id# = "D"
	AEB5	106	A=C S&X	
	AEB6	2A5	?NC XQ	
	AEB7	10C	->43A9	[CHKBFA]
NOTFUND	AEB8	3E0	RTN	
BFOUND	AEB9	05E	C=0 MS	delete only the first nybble
	AEBA	2F0	WRTDATA	so the OS will do the rest!
	AEBB	04E	C=0 ALL	
	AEBC	270	RAMSLCT	
	AEBD	051	?NC GO	Pack IO/KA area
	AEBE	086	->2114	[PKIOAS]

CHKBFA	43A9	0A6	A<>C S&X	recall id# to C(0)
CHKBF4	43AA	23C	RCR 2	id# to C(12)
	43AB	35C	PT= 12	
	43AC	130	LDI S&X	
	43AD	0BF	CON: 191	Fisrt possible reg -1
	43AE	10E	A=C ALL	store id# & addr in A
CB10	43AF	166	A=A+1 S&X	Increase reg# address
CB20	43B0	046	C=0 S&X	
	43B1	270	RAMSLCT	Select Chip 0
	43B2	378	READ 13(c)	.END.
	43B3	306	?A<C S&X	did we reach the .END. Chainhead?
	43B4	3A0	?NC RTN	yes -> Not Found
	43B5	0A6	A<>C S&X	addr to C[S&X]
	43B6	270	RAMSLCT	Candidate address for header
	43B7	0A6	A<>C S&X	id# to A(12) & addr to A[S&X]
	43B8	038	READATA	Candidate Value for header
	43B9	2EE	?C#0 ALL	Carry if not empty register
	43BA	3A0	?NC RTN	**empty reg -> Not Found**
	43BB	23E	C=C+1 MS	Carry if id#="F" (KAR)
	43BC	39F	JC -13d	Key Assignment Register
	43BD	362	?A#C @PT	is this IO Buffer?
	43BE	037	JC +06	NO , keep searching
	43BF	1B0	POPADR	YES !
	43C0	23A	C=C+1 M	Return to (P+2)
	43C1	170	PUSHADR	
	43C2	038	READATA	Return with Header in C
	43C3	3E0	RTN	and BuffAdr in A - rg# selected
CB30	43C4	0FC	RCR 10	Skip Buffer
	43C5	056	C=0 XS	
	43C6	146	A=A+C S&X	add buffer size
	43C7	34B	JNC -23d	[CB20]

4. Non-Linear Systems

Cubic Spline Interpolation (by Greg McClure)

Here is the authoritative implementation of cubic spline interpolation, using the extended functions to store the data points and the matrix function set from either the SandMatrix or the HP-41 Advantage indistinctly. This is the optimal arrangement for improved speed and reduced code size to resolve the problem.

The cubic spline algorithm is a mathematical interpolation method used to create a smooth curve between points on a graph. At least four data points are required to create a cubic spline. These data points need to be placed into a data array in Extended Memory. The user can decide the name of this array. After creating this array and filling it with X,Y data pairs, executing CSPLINE will create the splines needed to graph the curve. The program requires Matrix functions to create the solution, so either the Advantage or SandMatrix module is required for this program. If using the SandMatrix module, the SandMath module will also be required.

To solve the cubic spline, first a tri-diagonal matrix needs to be created, and a solution vector. "*ABC" and "*R" matrices are temporarily created for this purpose. Once solved, the "*MN" data array is created with the cubic spline coefficients, and "*ABC" and "*R" are removed. "*MN" is used to display the interpolated Y and YPRIME for any X entered.

It is easiest to show how to use **CSPLINE** by giving an example.

Let's say we want to create a smooth curve going thru points [0,0], [1,1], [2,9], and [3,10]. We wish to use the "natural" slope of the curve at both the beginning and the end (it can optionally be specified for either end).

Let's create data array "XY" (needs to be size 8). So "XY", 8, CRFLD. Now ensure flag 8 is off and XEQ **DFED**. Enter in points 0, 0, 1, 1, 2, 9, 3, and 10 at the prompts. We are now ready to perform the spline interpolation.

XEQ **CSPLINE**. It asks for the array name, enter XY and press R/S, it asks if we want initial slope. No response at this prompt forces "natural" slope for the initial point. R/S then asks for the the final slope. No response forces "natural" slope for the final point. R/S then displays the steps as it calculates the arrays, then solves the simultaneous equations created.

We are ready to display results for points. LBL A is always available to quickly get to this point. After the solution is created, we are at this label. R/S prompts for "NEXT X?", enter the X value we want Y value for. R/S calculates the Y coordinate interpolated, and R/S again calculates the slope at that point. In this example, let's get the slope calculated for the first point [0,0]. Enter 0, R/S, and it shows Y is 0 (not a surprise), R/S gives a slope of -1.3333. R/S (or A) and get "NEXT X?", 1, R/S and it shows Y is 1 (not a surprise), R/S gives a slope of 5.6667. X = 1.5 gives Y=5, and slope of 9.1667, and so on for any other points you want to solve. The natural slope at the end point of 3 was also -1.3333.

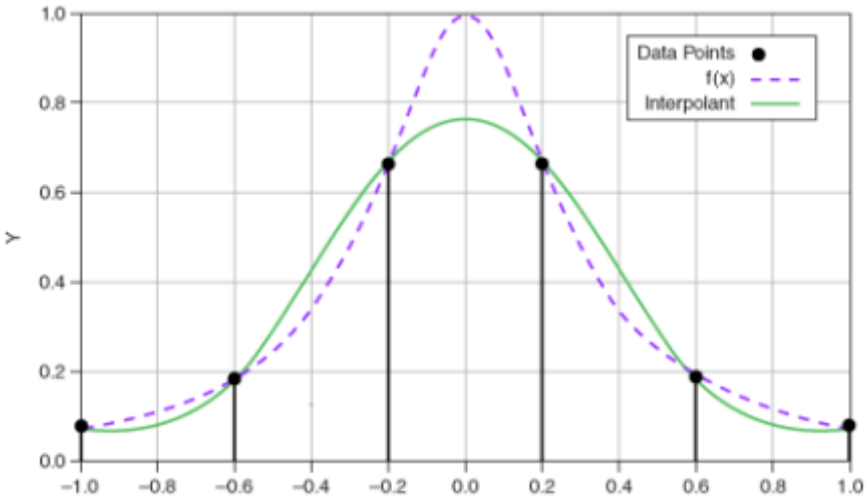
We can rerun the program and specify initial and final slopes if we wish, try it and see what points and slopes are interpolated with initial and final slopes of 0.

Example. Using the Data File Editor routine **DFED** make a data array (named "XY", with size 8) loaded with 1, 1, 2, 2, 3, 9, 4, 10 (this represents points [1,1], [2,2], [3,9] and [4,10]).

Run the program, specify "XY" as the array name, and when it asks for initial and final slope, just hit R/S (this tells the program to use natural slope ends). After it finishes (it says "READY"), hit R/S and enter the X coordinate you want to see the spline value for, it will display the Y coordinate, then you hit R/S to get the slope at that point.

- for x = 1.0, you should get: Y = 1.0, YP (slope)=-1
- for x = 1.5, you should get: Y=0.75, YP=0.5,
- for x = 2.5, you should get: Y=5.5, slope of 8.

As you can see, you can ask for any X points you want, and quickly make a sketch of the resulting spline curve fit for these points.



Here the "natural" spline slope is -1 for each end, and by specifying the desired slope for the ends you can change the shape of the resulting spline. To do that, simply rerun the CSPLINE program.

With 425 steps, **CSPLINE** is a large program, but the usability factors are well worth the price of admission: feedback of each step is provided during the execution so you know what progress is being made.

```
01  *LBL 10
02  "` SLOPE?"
03  CF 22
04  PROMPT
05  RTN
06  *LBL 99
07  SF 25
08  PURFL
09  CF 25
10  MATDIM
11  RTN
12  *LBL 96
13  "*CF"
14  RCL 09
15  INT
16  E
17  E3/E+
18  *
19  E-3
20  -
21  MSIJA
22  RTN
23  23*LBL 95
24  24"*R"
25  RCL 09
26  INT
27  MSIJA
28  RDN
29  RTN
30  30*LBL 94
31  DSE X
32  E3/E+
33  STO 09
34  RTN
35  LBL "CSPLINE
36  *LBL 00
37  CF 00
38  CF 01
39  "X,Y ARRAY?"
40  AON
41  PROMPT
42  AOFF
43  ASTO 00
44  FLSIZE
45  2
46  /
47  STO 09
48  48"*R"
49  XEQ 99
50  E
51  E3/E+
52  *
53  "*CF"
54  XEQ 99
55  CLA
56  ARCL 00
57  RCL 09
58  XEQ 94
59  CLX
60  SEEKPTA
61  "INIT"
62  XEQ 10
63  FC? 22
64  SF 00
65  FS? 22
66  STO 07
67  "FINAL"
68  XEQ 10
69  FC? 22
70  SF 01
71  FS? 22
72  STO 08
73  "MATRICES
    ..."
74  CF 21
75  AVIEW
76  GETX
77  STO 01
78  GETX
79  STO 02
80  GETX
81  STO 03
82  GETX
83  STO 04
84  "*CF"
85  2 E-3
86  MSIJA
87  FS? 00
88  GTO 00
89  CLX
90  GTO 01
91  *LBL 00
92  RCL 03
93  RCL 01
94  -
95  1/X
96  *LBL 01
97  MS
98  J-
99  FS? 00
100 GTO 00
101 E
102 GTO 01
103 *LBL 00
104 ST+ X
105 *LBL 01
106 MS
107 "*R"
```

```
108 E
109 MSIJA
110 RDN
111 FS? 00
112 GTO 00
113 RCL 07
114 GTO 01
115 *LBL 00
116 2
117 /
118 X^2
119 RCL 04
120 RCL 02
121 -
122 *
123 3
124 *
125 *LBL 01
126 MS
127 *LBL 98
128 ISG 09
129 GTO 00
130 GTO 97
131 *LBL 00
132 CLA
133 ARCL 00
134 FLSIZE
135 GETX
136 STO 05
137 GETX
138 STO 06
139 XEQ 96
140 RCL 03
141 RCL 01
142 -
143 1/X
144 MSR+
145 RCL 05
146 RCL 03
147 -
148 1/X
149 J+
150 MS
151 STO T
152 X<>Y
153 STO Z
154 +
155 ST+ X
156 J-
157 MS
158 RDN
159 XEQ 95
160 X^2
161 RCL 04
162 RCL 02
163 -
164 *
165 X<>Y
166 X^2
167 RCL 06
168 RCL 04
169 -
170 *
171 +
172 3
173 *
174 MS
175 RCL 03
176 STO 01
177 RCL 04
178 STO 02
179 RCL 05
180 STO 03
181 RCL 06
182 STO 04
183 GTO 98
184 *LBL 97
185 XEQ 96
186 FS? 01
187 GTO 00
188 CLX
189 GTO 01
190 *LBL 00
191 RCL 03
192 RCL 01
193 -
194 1/X
195 *LBL 01
196 MSR+
197 FS? 01
198 GTO 00
199 E
200 GTO 01
201 *LBL 00
202 ENTER^
203 ST+ X
204 *LBL 01
205 MS
206 X<>Y
207 XEQ 95
208 FS? 01
209 GTO 00
210 RCL 08
211 GTO 01
212 *LBL 00
213 X^2
214 RCL 04
215 RCL 02
216 -
217 *
218 3
219 *
220 *LBL 01
221 MS
222 "SOLVING"
223 AVIEW
224 "*CF,*R"
225 MSYS
226 "COEFFS..."
227 AVIEW
228 "*CF"
229 PURFL
230 RCL 09
231 INT
232 DSE X
233 ST+ X
234 "*MN"
235 SF 25
236 PURFL
237 CF 25
238 CRFLD
239 2
240 /
241 ISG X
242 ""
243 XEQ 94
244 CLA
245 ARCL 00
246 CLX
247 SEEKPTA
248 GETX
249 STO 01
250 GETX
251 STO 02
252 *LBL 93
253 XEQ 95
254 MRC+
255 STO 05
256 MR
257 STO 06
258 CLA
259 ARCL 00
260 FLSIZE
261 GETX
262 STO 03
263 GETX
264 STO 04
265 RCL 06
266 CHS
267 RCL 03
268 RCL 01
269 -
270 *
271 RCL 04
272 RCL 02
273 -
274 +
275 RCL 05
276 RCL 03
277 RCL 01
278 -
279 *
280 RCL 04
281 RCL 02
282 -
283 -
284 "*MN"
285 FLSIZE
286 RDN
287 SAVEX
288 X<>Y
289 SAVEX
290 RCL 03
291 STO 01
292 RCL 04
293 STO 02
294 ISG 09
295 GTO 93
296 SF 27
297 "READY"
298 PROMPT
299 *LBL A
300 "NEXT X?"
301 PROMPT
302 STO 07
303 CLA
304 ARCL 00
305 FLSIZE
306 2
307 /
308 DSE X
309 STO 09
310 CLX
311 SEEKPT
312 *LBL 92
313 GETX
314 RCL 07
315 X<Y?
316 GTO 00
317 GETX
318 DSE 09
319 GTO 92
320 RCLPT
321 ISG X
322 GTO 01
323 GTO 01
324 *LBL 00
325 RCLPT
326 *LBL 01
327 DSE X
328 GTO 00
329 GTO 01
330 *LBL 00
331 2
332 -
333 *LBL 01
334 CLA
335 ARCL 00
```

```
336  SEEKPTA
337  GETX
338  STO 01
339  GETX
340  STO 02
341  RCL Z
342  GETX
343  STO 03
344  GETX
345  STO 04
346  RCL Z
347  "*MN"
348  SEEKPTA
349  GETX
350  STO 05
351  GETX
352  STO 06
353  RCL 07
354  RCL 01
355  -
356  RCL 03
357  RCL 01
358  -
359  /
360  STO 08
361  E
362  -
363  CHS
364  STO 09
365  RCL 05
366  *
367  RCL 08
368  RCL 06
369  *
370  +
371  RCL 09
372  RCL 08
373  -
374  *
375  RCL 03
376  RCL 01
377  -
378  /
379  RCL 06
380  RCL 05
381  -
382  RCL 08
383  *
384  RCL 09
385  *
386  RCL 03
387  RCL 01
388  -
389  /
390  +
391  RCL 04
392  RCL 02
393  -
394  RCL 03
395  RCL 01
396  -
397  /
398  +
399  RCL 05
400  RCL 09
401  *
402  RCL 06
403  RCL 08
404  *
405  +
406  RCL 08
407  *
408  RCL 09
409  *
410  RCL 04
411  RCL 08
412  *
413  +
414  RCL 02
415  RCL 09
416  *
417  +
418  "Y="
419  ARCL X
420  PROMPT
421  "YP="
422  ARCL Y
423  PROMPT
424  GTO A
425  END

426  *LBL "DFED"
427   FLSIZE
428    E
429   -
430    E3
431   /
432  *LBL 00
433  SEEKPT
09   GETX
10   X<>Y
11   "D"
12   AINT
13   "|-="
14   X<>Y
15   ARCL X
16   CF 22
17   FC? 08
18   "|-?"
19   PROMPT
20   FC?C 22
21   GTO 02
22   FS? 08
23   GTO 01
24   X<>Y
25   RDN
26   X<>Y
27   SEEKPT
28   X<>Y
29   SAVEX
434  *LBL 02
30   X<>Y
435  *LBL 01
31   ISG X
32   GTO 00
33   "DONE"
34   AVIEW
35   CLA
36   FLNAME
37   END
```

Program **DFED** has been created to enter/review the data points in the array. First the data array must be created. For a 4-point solution the size must be 8, so with the name of the array in Alpha, and 8 in X, use **CRFLD** (use PURFL if already created). If user flag 8 is set, **DFED** will only view the data in the array. If user flag 8 is clear, it will show and optionally allow modification of the data in each element. If entering data, just enter each element until done.

Systems of Non-Linear Equations (Baillard - McClure)

This version is Greg McClure's direct modification using the Advantage Matrix functions of Jean-Marc Baillard's program to handle this problem, as documented on his web site:

http://hp41programs.yolasite.com/system-eq.php

The technique used is therefore exactly the same one used there: a (quasi-) Newton's method is used on each iteration to solve a linear system of n equations in n unknowns. The obvious difference in this version is the utilization of **MSYS** instead of Jean-Marc's "LS" routine, contributing to faster execution and reduced code size – while not introducing any restriction or limitation.

The user needs to program the "n" equations as independent FOCAL routines, with a global label each. With regard to the expected locations of the variables, it's of course impossible to take the n variables from the stack and calculate the n functions in the stack if n > 4, therefore you'll have to key in n different subroutines for computing the function in the X-register with x_1 in R01, x_2 in R02, ... ,and x_n in Rnn

Synthetic registers {M N O} and data registers R00 thru Rn^2+4n are used by the program. It also requires two initial guess-vectors (x1, x2,..xn) and (x1', x2',... xn') which components are to be stored into {R01- Rnn} and {Rnn+1 to R2n} respectively (and ensuring that $x_i \# x'_i$ for i = 1 , 2 , ... , n).

The successive x_1-values are displayed during the calculations, and when the program stops, $|f_1| + ... + |f_n|$ is in the X-register; and the solution (x_1 , ... , x_n) is in { R01 , ... , Rnn }.

The table below summarizes the data input requirements for "**NLSYS**":

Register	Value		Register	Value		Register	Value
R00	n						
R01	x1		Rnn+1	x1'		R2n+1	F1 Name
R02	x2		Rnn+2	x2'		R2n+2	F2 Name
.....	.....		.....	.....		.....	.....
Rnn	xn		R2n	xn'		R3n	Fn Name

All this manual data entering can be bothersome, therefore you'll be glad to know that the module includes several auxiliary routines to make the complete process more convenient. First off, the driver program **NLSN** will present all needed prompts for the input parameters automatically, storing them in the appropriate data registers. Within the driver program there are calls to other utilities to input the initial guesses (**XIN**) and the function names (**FIN**). Finally, after the system has been resolved, the driver program will invoke a data-output routine (**XOUT**) to show the results. All this will happen transparently to the user.

Example. program with the routines "**F1**", "**F2**", and "**F3**", defined :

f1(x,y,z) = x^2 + y – 3
f2(x,y,z) = y^2 - z -1
f3(x,y,z) = x - z^2 + 8

The three solutions are: x = 1, y = 2, z = 3

Go ahead and execute **NLSN**, using as initial guesses the values X0 = (1, 1, 1) andX0'= (2, 2, 2).

LBL "F1", RCL 01, X^2,RCL 02, +, -, RTN,
LBL "F2", RCL 02, X^2, RCL 03, -, 1, -, RTN,
LBL "F3", RCL 01, RCL 03, X^2, -, 8, +, RTN,

Here's the full data entry sequence with the driver program, Note that existing values will be suggested, jut press R/S to re-use them when appropriate:

XEQ "NLSN"	N= ?
4, R/S	F#1= . . . ?
"F1", R/S	F#2= . . . ?
"F2", R/S	F#3= . . . ?
"F3", R/S	X0'(1)= ?
2, R/S	X0'(2)= ?
2, R/S	X0'(3)= ?
2, R/S	X0(1)= ?
1, R/S	X0(2)= ?
1, R/S	X0(3)= ?
1, R/S	X3= 3.000000000
R/S	X2= 2.000000000
R/S	X1= 1.000000000

Example 2.Here is an example from Jean-Marc's documentation for a 4 element non-linear simultaneous equation set:

$$x1 + x2 + x3 + x4 - 16 = 0$$
$$x1.x2.x3 - 3.x4 = 0$$
$$4.x1^2 - x2.x3.x4 - 40 = 0$$
$$x1.x2.x3.x4 - 140 = 0$$

The following program should be entered by the user (assuming global labels chosen are F1, F2, F3, F4):

LBL "F1", RCL 01, RCL 02, RCL 03, RCL 04, +, +, +, 16, -, RTN,
LBL "F2", RCL 01, RCL 02, RCL 03, *, *, RCL 04, 3, *, -, RTN,
LBL "F3", RCL 01, ST+ X, X^2, RCL 02, RCL 03, RCL 04, *, *, -, 40, -, RTN,
LBL "F4", RCL 01, RCL 02, RCL 03, RCL 04, *, *, *, 140, -, END

The solutions and the locations of the values are as follows:

X1 = R01 = 4.266540475, X2 = R02 = 1.353632234
x3 = R03 = 3.548526784, x4 = R04 = 6.831300511

```
 1  *LBL "NLSN"
 2  SIZE?
 3  "N=?"
 4  PROMPT
 5  STO 00
 6  X^2
 7  LASTX
 8  4
 9  *
10  +
11   E
12  +
13  X>Y?
14  PSIZE
15  SF 01
16  XROM "FIN"
17  RCL 00
18  1.1
19  +
20  STO 02
21  RCL 00
22  E3/E+
23  STO 01
24  *LBL 01
25  "X0'("
26  RCL 01
27  AINT
28  "`)=?"
29  PROMPT
30  STO IND 02
31  ISG 02
32  ISG 01
33  GTO 01
34  XROM "XIN"
35  XROM "NLSYS"
36  *LBL "XOUT"
37  37*LBL 05
38  38"X"
39  RCL 00
40  AINT
41  41"`="
42  ARCL IND 00
43  PROMPT
44  DSE 00
45  GTO 05
46  END

47  *LBL 09
48   SF 25
49   PURFL
50   CF 25
51  MATDIM
52   RTN
53  *LBL "NLSYS"
54   RCL 00
55  "*S"
56   XEQ 09
57  .1
58  %
59  +
60  "*M"
61   XEQ 09
62  *LBL 99
63   VIEW 01
64  STO N
65  ST+ X
66  RCL 00
67  E3/E+
68  +
69  STO M
70  RCL N
71  +
72  STO N
73  STO O
74  *LBL 14
75   RCL IND M
76   XEQ IND X
77   STO IND N
78  ISG M
79  CLX
80  ISG N
81  GTO 14
82  RCL O
83  ENTER^
84  CLX
85  "*S"
86  MSIJA
87  RDN
88  STO M
89  STO N
90  *LBL 13
91  RCL IND M
92  MSC+
93  ISG M
94   GTO 13
95   RCL N
96   ENTER^
97   CLX
98   "*M"
99   MSIJA
100  RDN
101  STO N
102  RCL 00
103  E3/E+
104  STO M
105  STO O
106  *LBL 08
107  RCL IND M
108  RCL M
109  RCL 00
110  +
111  RCL IND X
112  STO IND M
113  RDN
114  X<>Y
115  STO IND Y
116  RTN
117  *LBL 12
118  XEQ 08
119  RCL N
120  RCL 00
121  -
122  RCL IND X
123  XEQ IND X
124  CHS
125  RCL IND N
126  +
127  MSR+
128  XEQ 08
129  ISG M
130  GTO 12
131  RCL O
132  STO M
133  ISG N
134  GTO 12
135   "*M,*S"
136  MSYS
137  "*S"
138  CLX
139  MSIJA
140  STO O
141  RCL 00
142  E3/E+
143  STO M
144  LASTX
145  .1
146  %
147  +
148  +
149  STO N
150  *LBL 11
151  RCL IND N
152  RCL IND M
153  STO IND N
154  X<>Y
155  -
156  ENTER^
157  ABS
158  ST+ O
159  RDN
160  MRC+
161  *
162  ST- IND M
163  ISG N
164  NOP
165  ISG M
166  GTO 11
167  RCL O
168  E-8
169  X>Y?
170  GTO 00
171  RCL 00
172  .1
173  %
174  +
175  GTO 99
176  *LBL 00
177  X<>Y
178  CLA
179  VIEW 01
180  END
```

```
1   *LBL "XIN"
2   RCL 00
3   E3/E+
4   CF 22
5   *LBL 02
6   "X0("
7   AINT
8   " `)="
9   ARCL IND X
10  "`?"
11  PROMPT
12  FS? 22
13  STO IND Y
14  FS?C 22
15  RDN
16  ISG X
17  GTO 02
18  RTN
19  *LBL "FIN"
20  RCL 00
21  FS? 01
22  ST+ X
23  1.1
24  +
25  STO 02
26  RCL 00
27  E3/E+
28  STO 01
29  AON
30  CF 23
31  *LBL 00
32  "F#"
33  RCL 01
34  AINT
35  "`? "
36  ARCL IND 02
37  STOP
38  FS?C 23
39  ASTO IND 02
40  ISG 02
41  ISG 01
42  GTO 00
43  AOFF
44  END
```

Example 3. Here's a more challenging one, with ten equations including trigonometric functions as And now is when we get to say the infamous words: "*the resolution is left to the reader as an exercise*…. "

$$2\,x_1 + x_1\,atan(x_2\text{-}x_{10}) + \cos x_7 x_8 - x_3\,x_4 - \tan x_8$$

$$x_2^2 exp(x_5.\,x_6) + exp(-\,x_8.x_9) - x_1\,x_7 - x_3 - x_9$$

$$sin(1\text{-}x_2\,x_{10}) + x_2.\,x_3.\,x_7 - (x_1.\,x_{10})^2 + \tan x_3 - \tan x_8$$

$$x_2\,x_{10} - x_8\,x_9 + \sin x_5 - x_4\,x_7^2$$

$$x_5.\,exp(x_8) - x_4\,\sin x_7 - \cos x_{10} - x_5$$

$$exp(\cos x_5) - exp(x_3) - x_1\,x_2\,x_{10} + x_6^2 + x_1$$

$$x_6\,x_7 + x_2\,x_9 - x_1\,\sin x_8 - x_4\,\sin x_7$$

$$atan(1\text{-}x_9) - \cos(x_3\text{-}x_6) - x_2\,x_5 + 2x_8$$

$$x_9.exp(x_4\,x_6) - \tan(x_2\,x_5) - x_2\,x_{10} + x_1\,x_9 - x_6$$

$$x_3\,x_8\,x_{10} - x_4\,x_7\,x_{10} - x_1\,x_{10} + x_2\,x_9$$

Solution: Set RAD mode, program the 10 equations andusingtheinitialguess:

[X1] = (1, 1, 1, 1, 1, 1, 1, 1, 1,1)

We obtain:

	X10=0.926763	;	X9=0.941375
	X8=0.950487	;	X7=0.842850
	X6=0.754390	;	X5=0.949426
	X4=1.213977	;	X3=0.878089
	X2=1.018692	;	X1=0.846165

Bessel J(x) via Continued Fractions Method - Martin-Baillard

The SandMath contains a very competent set of Bessel functions, both for the direct (J, Y) and the modified kinds (I, K). The implementation is a hybrid of MCODE and Focal routines, really optimized for the applicable valid range of the functions.

And therein lays the only caveat: that implementation does a direct sum of the alternating terms of the series, which isn't valid for asymptotic cases, where either the order or the argument (or their sum!) are very large. To palliate this, the SandMath also includes an iterative approach for JNX, using recurrence formulas – but alas, the execution time can be really long.

Is there another way to skin this cat? Well as it turns out yes, at least for the non-modified cases there's a very intriguing approach based on continued fractions, which after all are another way to iterate for the solution – only that we can take advantage of the MCODE implementation in both the SandMath and the 41Z Modules, because there are two different continued fractions involved, one of them in the complex variable – eve for the real Bessel J case!

Here again the routine is a direct modification of Jean-Marc Baillard's FOCAL program available on his web site (cf #5 in http://hp41programs.yolasite.com/bessel.php), adapted to use the MCODE functions **CF2V** and **ZCF2V**instead of the FOCAL subroutines – faster and shorter code. A real beauty to see the SandMath and 41Z joining forces to crack this one!

The formulas used are as follows:

With $p + i.q = -1/(2x) + i + (i/x) [(0.5^2 - n^2)/(2x + 2i + (1.5^2 - n^2)/(2x + 4i + ...))]$

and $g_n = -1/(((2n + 2)/x) - 1/(((2n + 4)/x) - .....))$

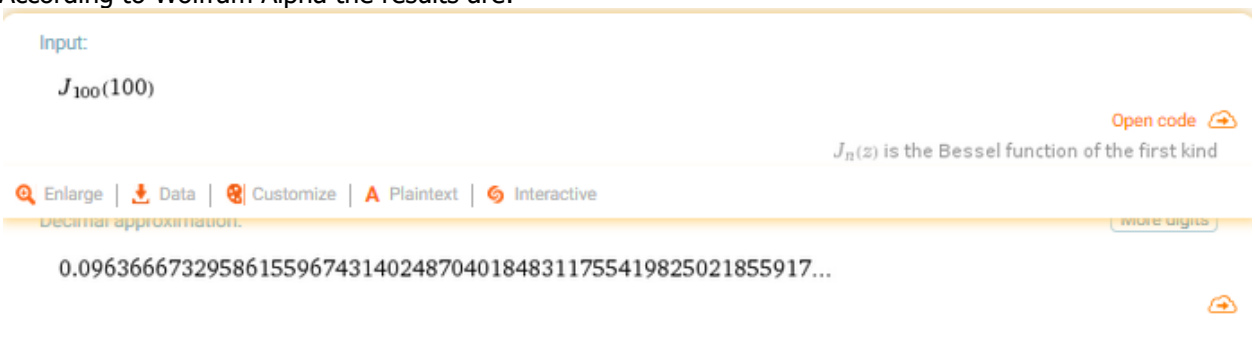
Then, calling D = the denominator of the second continued fraction:

$$J_n(x) = \text{sign}(D) [(2q/(x.Pi)) / (q^2 + (p - g_n - n/x)^2)]^{1/2}$$
$$Y_n(x) = [(p - g_n - n/x)/q] J_n(x)$$

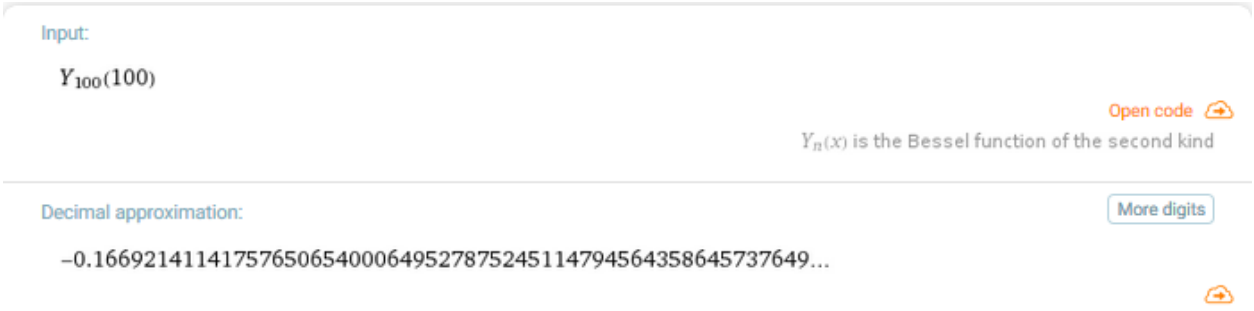
One must pay careful attention to the data registers requirements by these functions for the successions used to define the continued fractions, which are programmed under the global labels "#" for the real one and '=" for the complex one.

Example: Calculate the Bessel J and Y of order 100 for the argument x=100

According to Wolfram Alpha the results are:



and:



And sure enough this is what we obtain (with ten digit precision) using our routine:

100, ENTER^, XEQ "JYNX" => 0.09636667380
X<>Y -0.1669214116

Program Listing:

```
01  LBL "JYNX"
02  STO 01
03  X<>Y
04  STO 13
05  "="
06  CLST
07  ZENTER^
08  .
09  RCL 01
10  ZCF2V
11  RCL 02
12  STO 01
13  ST/ Z
14  /
15  E
16  +
17  STO 10
18  X<>Y
19  CHS
20  RCL 01
21  ST+ X
22  1/X
23  -
24  STO 09
25  "#"
26  0
27  RCL 01
28  CF2V
29  CHS
30  RCL 09
31  +
32  RCL 13
33  RCL 01
34  /
35  -
36  STO 11
37  RCL 10
38  R-P
39  LASTX
40  ST+ X
41  PI
42  RCL 01
43  *
44  /
45  SQRT
46  X<>Y
47  /
48  RCL 05
49  SIGN
50  *
51  STO 12
52  RCL 11
53  *
54  RCL 10
55  /
56  RCL 12
57  CLD
58  RTN
59  LBL "="
60  RCL 12
61  ST+ X
62  RCL 02
63  ST+ X
64  ZENTER^
65  RCL 12
66  660.5
67  -
68  X^2
69  RCL 13
70  X^2
71  -
72  0
73  X<>Y
74  RTN
75  LBL "#"
76  X<>Y
77  STO 05
78  X<>Y
79  RCL 02
80  RCL 13
81  +
82  ST+ X
83  RCL 01
84  /
85  -1
86  END
```

Note: ensure that the ADVTG_MATH module is plugged in a page before the SandMath. This is required because there is another global label "=" in the SandMath and we don't want the routine to use the incorrect one for the calculation! (besides, this would result in NONEXISTENT, so you'll know right away).

Newton's and Halley's Methods Revisited - Martin-McClure

The idea of using the MCODE functions in the SandMath is also at the heart of this final application, as this time we'll use the first &second derivatives function **DERV** as an auxiliary tool to calculate the derivatives of the function whose roots we're trying to obtain, directly and without any additional conditioning regardless of the function in case.

The formulas involved are well known:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \quad ; \quad x_{n+1} = x_n - \frac{2f(x_n)f'(x_n)}{2[f'(x_n)]^2 - f(x_n)f''(x_n)}$$

As usual, you need to provide two guesses in the X,Y registers and the function name in ALPHA. The user is required to program the function in a FOCAL routine under a global label, which cannot use data registers R00 to R08 as explained below.

Remember that **DERV** uses R00 to R04 (see the documentation in the SandMath manual for details), and in addition to these the routines use R05 for the function global label name, and R06 – R08 to save the initial guesses and as scratch. As it's already customary, the successive approximations to the root will be displayed if user flag 10 is set.

```
01  *LBL "XNWT"
02  CF 01
03  GTO 01
04  *LBL "XHALL"
05  SF 01
06  *LBL 01
07  ASTO 05
08  X<>Y
09  STO 08
10  X<>Y
11  *LBL 00
12  FS? 10
13  VIEW X
14  STO 06
15  XEQ IND 05
16  STO 07
17  RCL 08
18  RCL 06
19  DERV
20  FC? 01
21  ST/ 07
22  FS? 01
23  XEQ 02
24  RCL 06
25  RCL 06
26  RCL 07
27  -
28  X#Y?
29  GTO 00
30  CLD
31  RTN
32  *LBL 02
33  X^2
34  ST+ X
35  RCL 07
36  RCL 01
37  *
38  -
39  1/X
40  RCL 07
41  *
42  RCL 00
43  *
44  ST+ X
45  STO 07
46  END
```

This really can't get any shorter; my kinda routine that clearly showcases that with a powerful engine behind doing the heavy lifting (**DERV** in this case) the rest is a downhill trip.

Example: obtain a root for the equation below, which we program easily as shown. Then we usesomeobviouslynon-optimal guesses to stress the algorithm:

$$y = \sqrt[3]{x}(x+4)$$

{ LBL "X1", CBRT, LASTX, 4, +, *, END }, and then

ALPHA,"X1", ALPHA, 1, 2, XEQ "XNWT" => -4.00000000

Or:` 1, 2, XEQ "XHALL" =>-4.00000000

Newton's Method with Complex Step Differentiation.

And the proverbial last but not least is reserved for the "complex step derivative" method to calculate real function derivatives, just as a quasi-magical application of complex variables. Complex step differentiation is a technique that employs complex arithmetic to obtain the numerical value of the first derivative of a real valued analytic function of a real variable, avoiding the loss of precision inherent in traditional finite differences. This is then used n Newton's method in the usual way.

We're concerned with an *analytic* function. Mathematically, that means the function is infinitely differentiable and can be smoothly extended into the complex plane. Computationally, it probably means that it is defined by a single "one line" formula, not a more extensive piece of code with if statements and for loops.

Let $F(z)$ be such a function, let $x0$ be a point on the real axis, and let h be a real parameter. Expand $F(z)$ in a Taylor series off the real axis.

$$F(x_0+ih)=F(x_0)+i.hF'(x_0)-h^2F''(x_0)/2! - ih^3F^{(3)}/3!+...$$

Take the imaginary part of both sides and divide by h

. $$F'(x_0)=Im(F(x_0+ih))/h+O(h_2)$$

Armed with the 41Z arsenal of functions it's very likely that your real function can be programmed as an equation in the complex variable too. Then all it takes is to calculate the value of said complex function in a complex point close to the real argument $x0$, offset by a very small amount in the imaginary axis ih. The program expects the program name in ALPHA and the values of h and x0 in the Y,X stack registers, and it returns the real derivative value in X. it uses data registers R00 to R02.

```
1    LBL "ZNWT"
2     ASTO 02
3     ZSTO
4     LBL 00
5    FS? 10
6    VIEW 00
7    ZRCL  (00)
8    XEQ IND 02
9    X<>Y
10   /
11   RCL 01
12   *
13   ST- 00
14   RND
15   X#0?
16   GTO 00
17   RCL 00
18   END
```

What's remarkable is that with just one execution of the complex function we calculate both the real function's value (the real part) and its derivative (the imaginary part with correction) at the same time. Note also the clever use of complex data register C00 to store z0 = x0 +ih, and then how it keeps calculating the complex function value until two successive iterations are equal for the current FIX selected in the calculator.

Something's remarkable when the root-finding routine is almost shorter than the equation use to program the function!

Time for some examples. The first one just a simple polynomial to try our hand with the new method, taken from the MoHPC forum: https://www.hpmuseum.org/forum/thread-6667.html

Calculate the three roots of the third degree polynomial: $x^3-x^2-x+0,5 = 0$

We program the equation as shown below:

```
01LBL "Z3"          06 Z-
02 Z^3              07 .5
03 LASTZ            08 +
04 Z^2              09 END
05 Z+
```

And type:

```
ALPHA, "Z1", ALPHA
,01, ENTER^, 0, XEQ "ZNWT"      =>  0.40301587
.01, ENTER^, 2, XEQ "ZNWT"      =>  1.45174468
.01, ENTER^, -2, XEQ "ZNWT"     =>  -0.85476055
```

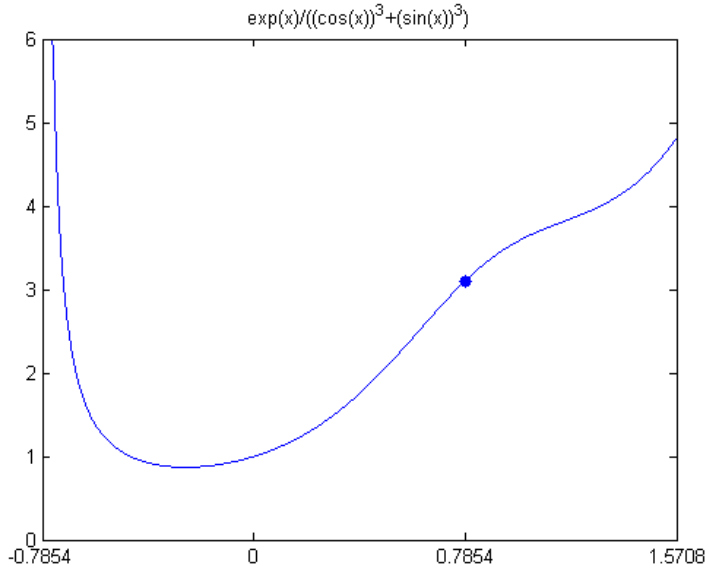
And then a more elaborate example adapted from the seminal reference:
https://blogs.mathworks.com/cleve/2013/10/14/complex-step-differentiation/

The blog uses the function F(x) given below, which does not have any real roots:

$$F(x) = \frac{e^x}{(\cos x)^3 + (\sin x)^3}$$

For our purposes let's calculate the roots of, say g(x) = F(x) – π

```
01 LBL "Z2"
02 ZEXP
03 LASTZ
04 ZSIN
05 LASTZ
06 ZCOS
07 3
08 Z^X
09 Z<>W
10 3
11 Z^X
12 Z+
13 Z/
14 PI
15 -
16 END
```



And type:

```
ALPHA, "Z2", ALPHA
,01, ENTER^, 1, XEQ "ZNWT"      =>0.79830245
```

Appendix: From Poles to Zeros. { **POLZER** }

This program completes the applications section. It calculates the zeros of a polynomial expressed as a partial expansion of factors, as would typically be the case when working with transfer functions in control theory.

This program calculates the polynomial coefficients and roots of expressions such as:

P(x) = Σ [1 / (x-pi)] ;i= 1,2,… n , for n<= 7

Which will be transformed into:

P(x) = Σai.x^i ;i= 0,1,… (n-1)

The coefficients are obtained using the following formulae:

a(n-1) = n
a(n-2) = (n-1) Σ pi
a(n-3) = (n-2) Σ Σ pi pj
a(n-4) = (n-3) Σ Σ Σ pi pj pk
a(n-5) = (n-4) Σ Σ Σ Σ pi pj pk pl
a(n-6) = (n-5) Σ Σ Σ Σ Σ pi pj pk pl pm

in general the n-th. coefficient would require the calculation of n-dimensional product sums. However the program **POLZER** is limited to expressions up to 7 poles max (resulting in 6 zeroes).

Example.- To study the stability of the transfer function below, calculate its roots.

G(s) = 1/s + 1/(s-1) + 1/(s-2) + 1/(s-3) + 1/(s-4)

Keystrokes	**Display**
XEQ "**POLZER**"	#POL :?
5, R/S	P(1) :?
0, R/S	P(2) :?
1, R/S	P(3) :?
2, R/S	P(4) :?
3, R/S	P(5) :?
4. R/S	Σ … ΣΣ .. ΣΣΣ… ΣΣΣΣ…. ΣΣΣΣΣ.
	CF57 Y/N
"**Y**"	a(4) : 5
R/S	a(3) : -40
R/S	a(2) : 105
R/S	a(1) : -100
R/S	a(0) : 24

Therefore the "natural" polynomial form is as follows:

G(s) = 5 s^4 – 40 s^3 + 105 s^2 – 100 s + 24

Next the execution is transferred to **PROOT** in the SandMatrix (or to **QUART** if #p=5) which calculates the roots following the iterative process explained in section 4.3.1. Remember that the accuracy is dictated by the number of decimals places set .

R/S RUNNING...
R/S x1= 3,64442
R/S x2= 2,54395
R/S x3= 1,45609
R/S x4= 0,35557

POLZER is also a rather long program – and dates back to the days the author attended EE School many moons ago, so I'm somehow attached to it.

Program Listing.

```
LBL "POLZER"
  RAD
  SIZE?
  23
  X>Y?
  PSIZE
LBL A
  CLRG
  8
  "#POL=?"
  PROMPT
  X>=Y?
  GTO A
  X>0?
  X=1?
  GTO A
  STO 00
  ,
  X<>F
  X<>Y
  SF IND X
  E3/E+
  STO 07
*LBL 00
  "P("
  RCL 07
  AINT
  "`)=?"
  PROMPT
  STO IND 07
  ISG 07
  GTO 00
  RCL 00
  STO 07
  ,
  STO 00
  CLA
  XEQ F
  RGSUM
  RCL 07
  E
  -
  *
  CHS
  STO 08
  FS? 02
  GTO 90
  XEQ F
  STO 14
  XEQ 04
  RCL 07
  2
  -
  *
  STO 09
  FS? 03
  GTO 90
  XEQ F
  STO 15
  XEQ 03
  RCL 07
  3
  -
  *
  CHS
  STO 10
  FS? 04
  GTO 90
  XEQ F
  STO 16
  XEQ 02
  RCL 07
  4
  -
  *
  STO 11
  FS? 05
  GTO 90
  XEQ F
  STO 17
*LBL 01
  RCL 07
  83 3
  -
  E3/E+
  RCL 17
  INT
  +
  STO 16
  ,
  STO 21
  XEQ 02
  RCL IND 17
  *
  ST+ 22
  ISG 17
  GTO 01
  RCL 22
  RCL 07
  5
  -
  *
  CHS
  STO 12
*LBL 90
  RCL 07
  E
  -
  STO 00
  TONE 0
  "CFS? Y/N"
  AVIEW
  sF#
  99
  CLX
  X#0?
  GTO 08
  RCL 00
  7
  +
  I<>J
  7
  +
  SF 21
  PVIEW
  CF 21
*LBL 08
-HL MATH+
  FS? 05
  GTO 14
  FS? 04
  GTO 13
  7
  RCL 07
  +
  E3/E+
  6
  +
  3
  PCPY
  PROOT
  GTO A
*LBL 14
  RCL 07
  ST/ 08
  ST/ 09
  ST/ 10
  ST/ 11
  RCL 08
  RCL 09
  RCL 10
  RCL 11
  QUART
  GTO 00
*LBL 13
  RCL 07
  RCL 08
  RCL 09
  RCL 10
```

```
160 CROOT
161 ΣV#
162  43
163  GTO 00
164 *LBL F
165  CF 21
166  "`s"
167  AVIEW
168  SF 25
169  SF 99
170  RCL 07
171  RCL 00
172  -
173  E3/E+
174  ISG 00
175  ""
176  RTN
177 *LBL 04
178  RCL 07
179 E3/E+
180  RCL 14
181  INT
182  +
183 RGSUM
184 RCL IND 14
185  *
186  ST+ 19
187  ISG 14
188  GTO 04
189 RCL 19
190  RTN
191 *LBL 03
192  RCL 07
193  E
194  -
195 E3/E+
196  RCL 15
197  INT
198  +
199  STO 14
200  ,
201  STO 19
202  XEQ 04
203  RCL IND 15
204  *
205  ST+ 20
206  ISG 15
207  GTO 03
208  RCL 20
209  RTN
210 *LBL 02
211 RCL 07
212  2
213  -
214 E3/E+
215  RCL 16
216  INT
217  +
218  STO 15
219  ,
220  STO 20
221  XEQ 03
222  RCL IND 16
223  *
224  ST+ 21
225  ISG 16
226  GTO 02
227  RCL 21
228  END
```