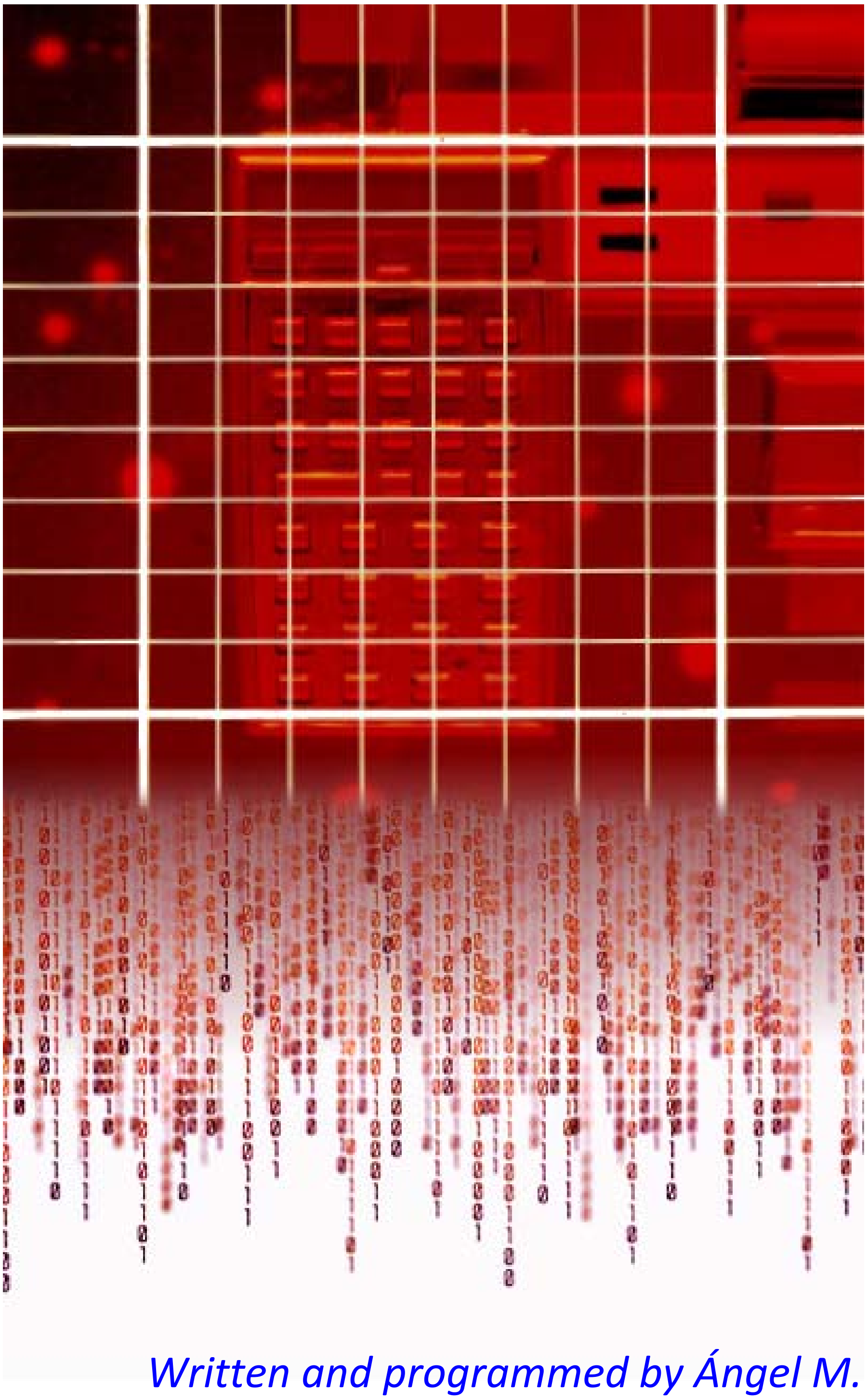
SANDMATRIX_4

Matrix Extensions for the HP-41.

User's Manual and Quick Reference Guide



Written and programmed by Ángel M. Martin

August 2013

This compilation revision 3.4.5

Acknowledgments.-

Documentation wise, this manual begs, steals and borrows from many other sources – in particular from the HP-41 Advantage Manual. Not so from the CCD Manual but obviously that was how it all began – with the excellent implementation of the Array Functions by W&W GmbH.

Thanks to the following contributors must be given: Jean-Marc Baillard; Valentín Albillo; Eugenio Úbeda; and Ulrich Deiters. Original authors retain all copyrights, and should be mentioned in writing by any party utilizing this material. No commercial usage of any kind is allowed.

Screen captures taken from V41, Windows-based emulator developed by Warren Furlow. Its breakpoints capability and MCODE trace console are a godsend to programmers. See www.hp41.org

Table of Contents. - Revision K.

Note: Make sure that revision "H" (or higher) of the Library#4 module is installed.

3. Upper Page Functions in Detail

SandMatrix_4 Module – Revision K
Matrix Extensions for the HP-41 System.

1. Introduction.

The release of the CCD Module by W&W in 1983 provided convenient and reliable tools for matrix algebra in the 41 platform for the first time. It was an MCODE quantum leap ahead, beyond the very many user programs written on the subject in the previous years. Looking back it's clear that the "ARRAY FNS" was beyond a doubt an amazing landmark in the legacy of the 41 platform. So much so that rather than re-invent the wheel HP decided to use it almost in its entirety in the Advantage Pac, only enhancing it with the major matrix operations sorely missing in the CCD implementation (which incidentally were the subject of the majority of Matrix programs written for the CCD).

Perhaps because the relative tardiness of its appearance, with the HP-42S already on the horizon - or due to other factors like the HP-48S luring folks into RPL - the fact is that Matrix programs using the Advantage Pac functions were very few and far in between. The demise of PPC and the newsletter wars that followed suit certainly didn't encourage the scene either, and the end result was slightly disappointing in terns of net results.

About 30 years later the SandMatrix picks up the gauntlet and compiles a collection of noteworthy programs and routines on Matrix and Polynomial algebra, with the specific criteria to be based on the CCD/Advantage function set – in an attempt to straighten the record and pay the due credit to that superb toolset that had been so underutilized.

1.1. The logical next chapter after the SandMath

In many respects the SandMatrix is a very conventional module. There are no fancy overlays or alternate keyboards, no auxiliary FATs with sub-functions, nor will you find dedicated function launchers á la SandMath. Most of the new routines are written in FOCAL, and the programs are typically large ones. Programming with the Matrix functions is more about Alpha strings and auxiliary data sets than concerning with data registers and to some extent even algorithmic strategy. Also because they are FOCAL programs they are slower than other areas, although the 41CL has blurred the lines separating MCODE and FOCAL in terms of speed.

In terms of its contents, it was clear from the beginning that it should be an extension to the SandMath. However the dilemma was how to manage the dependencies: should it be a self-contained ROM or rely on functions from other modules? The former option implied including many auxiliary functions in the FAT's, taking precious entries and causing redundancy in the global scheme. The latter option however meant a potential loss of usability, since several modules were involved – the Library #4, the SandMath, AMC_OS/X, the Solve & Integrate ROM, the Polynomial ROM, etc.

The solution to this riddle came only with the latest revision of the SandMath 3x3, which added a third bank with Solve and Integrate – plus an important consolidation of functions in its auxiliary FAT. This really cleared things off for the SandMatrix, in that the only dependencies left are the Library#4 and the SandMath itself – for a total of only 8k "effective" footprint needed additionally (since the Library#4 is located in the otherwise reserved page-4).

So there you have it, the SandMatrix more or less replaces all previous versions of the "Advanced Matrix ROM", the "Matrix ROM", and the "Polynomial ROM" (not counting the one co-produced w/ JM Baillard. Also in this regard it's worth mentioning that the SandMatrix is totally independent from the "JMB_Matrix ROM", which doesn't use the Advantage function set at all).

1.2. The many names of Dr. Who.

The SandMatrix is the last incarnation of a series of different modules previously released that also dealt with Matrix and Polynomial algebra. Some of them were based on the Advantage itself, combining the matrix functions with other applications and thus followed the same bank-switching implementation: two pages, with two banks in the upper page. The differences amongst them were about what else (beyond the matrix set) they included – once you removed the less notorious content of the Advantage.

The table below illustrates this, showing the dependencies and choices made in all the predecessors of the SandMatrix.

Size	Main	Dependency	Requires	Notes
8k + 8k	ALGEBRA	Advantage	n/a	
4k + 8k 4k	MATRIX_4k POLYN_4k	Advantage	n/a	
4k +8k	MATRIX_4L4	Advantage	Lib#4	
8k + 4k	Adv_Matrix	POLYN_4k	n/a	*Includes SOLVE/INTEG*
8k + 4k	Adv_Matrx4_I	POLYN_4L4	Lib#4	*Includes SOLVE/INTEG*
9k	Adv_Matrix4_II	n/a SIROM (*)	Lib#4	*Includes CURVE FIT* *(*) for EIGEN only*
8k	**SandMatrix**	**SandMath**	Lib#4	

We sure have a much simpler situation now, glad to say we left all those behind.

What isn't included?'

When compared to the original Advantage Pac, the following functionality areas are not included in the SandMatrix – but in other dedicated modules (and in a superior implementation if I may add), as shown in the table below:

Section	In Module	Also Available in	Comments
Digital Functions	Digit Pac	HP-IL Development	Includes 16C Emulator
Solve & Integrate	SandMath 3x3	Solve & Integrate ROM	Fully embedded
Curve Fitting	SandMath 3x3	AECROM	Fully embedded
Complex Operations	HP-41Z	-	Dedicated 8k ROM
Vectors / Coordinates	Vector Calculator ROM	-	Dedicated 4k ROM
Differential Equations	Diffeq ROM	Math Pac	Dedicated 8k ROM
Time Value of Money	Financial Pacs	HP-12C	don't care that much

Note: Make sure that revision "H" (or higher) of the Library#4 module is installed.

Function index at a glance.

And without further ado, here's the list of functions included in the module.

#	Function	Description	Input	Output	Author
1	-SNDMTRX 4	*Section Header*	*none*	*Displays "Order=?"*	*Ángel Martin*
2	ABSP	Alpha Back Space	Text in Alpha	Last char deleted	*W&W GmbH*
3	AIP	Appends integer part	x in X	INT(x) appended to Alpha	*Ángel Martin*
4	ASWAP	Alpha Swap	A,B in Alpha	B,A in Alpha	*Ángel Martin*
5	CLAC	CLA from Comma	Text in Alpha	Removed from left to comma	*W&W GmbH*
6	DOTN	N-dimensional Dot product	cntl words in Y,X	cntl word result in X	*JM Baillard*
7	EQT	Displays Curve Equation	Eq# in R00	Writes equation in Alpha	*Ángel Martin*
8	SQR?	Tests for Square matrices	Mname in Alpha	Yes.No – Do it true	*Ángel Martin*
9	*"MATRX"*	"Easy Matrix" Program	Driver for Major Matrix Ops.	Under prgm control	*HP Co.*
10	MPOL	Matrix polynomial	Mname in Alpha, Cnt'l word in X	Calculates P([A])	*Ángel Martin*
11	ST<>A	Swaps Alpha/Stack	V1 in Stack, V2 in Alpha	V2 in Stack, V1 in Alpha	*Ángel Martin*
12	V*V	3-dimensional Dot product	prompts for coeffs	result in Matrix	*Ángel Martin*
13	*"3DV"*	3D Vectors	Promppts "|V| V* VX"	performs operation	*Ángel Martin*
14	-CCD MTRX	*Section Header*	*none*	*Displays "Running..."*	*Ángel Martin*
15	C<>C	Column exchange (k<>l)	kkk,lll in X	Columns swapped	*W&W GmbH*
16	CMAX	Column Maximum	Col# in X, "OP1" in Alpha	Element value in X	*W&W GmbH*
17	CNRM	Column Norm	Col# in X, "OP1" in Alpha	colum norm in X	*W&W GmbH*
18	CSUM	Column Sum	"OP1,RES" in Alpha	Sum of Cols in RES matrix	*W&W GmbH*
19	DIM?	Matrix Dimension	"OP1" in Alpha	dimension placed in X	*W&W GmbH*
20	FNRM	Frobenius Norm	"OP1" in Alpha	value in X	*W&W GmbH*
21	I+	Increase row index	"OP1" in Alpha	increased i	*HP Co.*
22	I-	Decrease row index	"OP1" in Alpha	decreased i	*HP Co.*
23	J+	Increase column index	"OP1" in Alpha	increased j	*HP Co.*
24	J-	Decrease column index	"OP1" in Alpha	decreased j	*HP Co.*
25	M*M	Matrix Product	"OP1,OP2, RES" in Alpha	matrix product in RES	*W&W GmbH*
26	MAT*	element multiplication	value in X, "OP1,X" in Alpha	aij = aij * x	*W&W GmbH*
27	MAT+	addition of scalar	value in X, "OP1,X" in Alpha	aij = aij + x	*W&W GmbH*
28	MAT-	element substraction	value in X, "OP1,X" in Alpha	aij = aij - x	*W&W GmbH*
29	MAT/	Division by scalar	value in X, "OP1,X" in Alpha	aij = aij / x	*W&W GmbH*
30	MATDIM	Dimensions a matrix	mmm,nnn in X, "OP1" in Alpha	Matrix Dimensioned	*W&W GmbH*
31	MAX	Maximum element	"OP1" in Alpha	Element value in X	*W&W GmbH*
32	MAXAB	Absolute maximum	"OP1" in Alpha	Element value in X	*W&W GmbH*
33	MDET	Determinant	"OP1" in Alpha	Determinant in X	*HP Co.*
34	MIN	Minimum element	"OP1" in Alpha	minimum element in X	*W&W GmbH*
35	MINV	Inverse Matrix	"OP1" in Alpha	Matrix replaced w/ Inverse	*HP Co.*
36	MMOVE	Moves part of a matrix	I,j; k,l; b,m,n in XYZ	Elements moved	*W&W GmbH*
37	MNAME	Get current Mname to Alpha	none	Matrix Name in Alpha	*W&W GmbH*
38	MR	Recall element from pt	none	value in X	*HP Co.*
39	MRC+	Recall and advance in Column	"OP1" in Alpha	element in X, increased i	*W&W GmbH*
40	MRC-	Recall and back one in Column	"OP1" in Alpha	element in X, decreased i	*W&W GmbH*
41	MRIJ	Recall ij pointer of current	none	pointer in X	*W&W GmbH*
42	MRIJA	Recall ij pointer of Alpha	"OP1" in Alpha	pointer in X	*W&W GmbH*
43	MRR+	Recall and advance in Row	"OP1" in Alpha	element in X, increased j	*W&W GmbH*
44	MRR-	Recall and back one in Row	"OP1" in Alpha	element in X, decreased j	*W&W GmbH*
45	MS	Store element at pointer	value in X, OP1 in Alpha	Element stored	*HP Co.*
46	MSC+	Store and advance in Column	value in X, OP1 in Alpha	element stored, increased i	*W&W GmbH*
47	MSIJ	Sets pointer of current matrix	iii,jjj in X	pointer set	*W&W GmbH*
48	MSIJA	Sets points of Matrix in Alpha	iii,jjj in X; OP1 in Alpha	pointer set	*W&W GmbH*
49	MSR+	Store and advance in Row	value in X, OP1 in Alpha	element stored, increased j	*W&W GmbH*
50	MSWAP	Swapps part of a matrix	I,j; k,l; b,m,n in XYZ	Elements Swapped	*W&W GmbH*
51	MSYS	Linear Systems	"OP1,OP2, RES" in Alpha	Resolves Linear System	*HP Co.*
52	PIV	Sets pointer to pivot element	Col# in X, "OP1" in Alpha	Element value in X	*W&W GmbH*

#	Function	Description	Input	Output	Author
53	R<>R	Row Exchange (k<>l)	kkk,lll in X	Rows swapped	*W&W GmbH*
54	R>R?	Row comparison test	kkk,lll in X	skip line if false	*W&W GmbH*
55	RMAXAB	Absolute maximum	row# in X, OP1 in Alpha	element in X, pointer to ij	*W&W GmbH*
56	RNRM	Row Norm	"OP1" in Alpha	Row Norm in X	*W&W GmbH*
57	RSUM	Row Sum	"OP1,RES" in Alpha	sums of rows in RES matrix	*W&W GmbH*
58	SUM	Element Sum	"OP1" in Alpha	element sum in X	*W&W GmbH*
59	SUMAB	Absolute Values Sum	"OP1" in Alpha	element absolute sum in X	*W&W GmbH*
60	TRNPS	Transpose	"OP1" in Alpha	Matrix replaced w/ transposed	*HP Co.*
61	YC+C	Adds Y*Col (l) to Col (k)	value in Y, kkk.lll in X	column k changed	*W&W GmbH*
62	*"MEDIT"*	Matrix Editor	prompts for elements	Edits Matrix	*HP Co.*
63	*"CMEDIT"*	Complex Matrix Editor	prompts for coeffs	Edits Complex matrix	*HP Co.*
64	MPT	Matrix Prompt	Iii,jjj in x	Prompts for element	*Ángel Martin*
1	-ADV MATRIX	*Section Header*	*none*	*Displays "Not Square"*	*Ángel Martin*
2	^MROW	Input Row	"OP1" in Alpha, row# in x	Prompts for Row	*Ángel Martin*
3	I<>J	Swaps indexes	iii,jjj in X	j,00i in X, i00j in LastX	*Ángel Martin*
4	I#J?	Is i # j?	iii,jjj in X	comparison, skip if False	*Ángel Martin*
5	IMC	Input Matrix by Columns	"OP1" in Alpha	Inputs elements by columns	*Ángel Martin*
6	IMR	Input Matrix by Rows	"OP1" in Alpha	Inputs elements by rows	*Ángel Martin*
7	LU?	Tests for L/U Decomposed	Mname in Alpha	Yes.No – Do it true	*Ángel Martin*
8	M^1/X	x-th. root of a Matrix	"OP1" in Alpha, x in X	Matrix replaced by its root	*Ángel Martin*
9	M^2	Matrix Square	"OP1" in Alpha	Matrix replaced by [M][M]	*Ángel Martin*
10	MAT=	Copy Matrix	"OP1,RES" in Alpha	Copies matrix A into B	*Ángel Martin*
11	MATP	Driver for M*M	Driver for M*M	Under prgm control	*Ángel Martin*
12	MCON	Constant	"OP1" in Alpha, x in X	Makes all elements =x	*Ángel Martin*
13	MDPS	Diagonal Product Sum	"OP1" in Alpha	Sum of diagonal products	*Ángel Martin*
14	*"MEXP"*	Matrix Exponential	"OP1" in Alpha	Matrix replaced by exp(M)	*Ángel Martin*
15	MFIND	Element finder	"OP1" in Alpha, x in X	Element pointer if found	*Ángel Martin*
16	MIDN	Identity Matrix	"OP1" in Alpha	Makes it Identity Matrix	*Ángel Martin*
17	MLIE	Matrix Lie Product	"OP1,OP2,RES" in Alpha	[A][B] - [B][A]	*Ángel Martin*
18	MLN	Matrix Natural Log	"OP1" in Alpha	Matrix replaced by LN(M)	*Ángel Martin*
19	MPWR	Matrix Power to X	"OP1" in Alpha, x in X	Matrix replaced by [M]^INT(x)	*Ángel Martin*
20	MRDIM	Matrix Redimension	"OP1" in Alpha, dim in X	Matrix redimensioned	*Ángel Martin*
21	MSQRT	Matrix Square Root	"OP1" in Alpha	Matrix replaced by SQRT([M])	*Ángel Martin*
22	MSORT	Sorts matrix elements	"OP1" in Alpha	Matrix Elements sorted	*Ángel Martin*
23	MSZE?	Matriz Size	"OP1" in Alpha	Matrix size in X	*Ángel Martin*
24	MTRACE	Matrix Trace	"OP1" in Alpha	Trace in x	*Ángel Martin*
25	MZERO	Zeroes a Matrix	"OP1" in Alpha	All elements zeroed	*Ángel Martin*
26	OMC	Output Matrix by Columns	"OP1" in Alpha	Shows elements by columns	*Ángel Martin*
27	OMR	Output Matrix by Rows	"OP1" in Alpha	Shows elements by rows	*Ángel Martin*
28	OCX	Output x-th column	"OP1" in Alpha, Col# in X	Shows Col elements	*Ángel Martin*
29	ORX	Output x-th row	"OP1" in Alpha, Row# in X	Shows Row elements	*Ángel Martin*
30	PMTM	Prompts for Matrix	"OP1" in Alpha	Prompts for complete Rows	*Ángel Martin*
31	R/aRR	Unitary Diagonal	"OP1" in Alpha	Diagonal elements = 1	*Ángel Martin*
32	ΣIJJI	Sum of crossed products	"OP1" in Alpha	Σ[aij*aji] in X	*Ángel Martin*
33	-ADV POLYN	*Section Header*	*none*	*Displays "Σ(ak*X^k)"*	*Ángel Martin*
34	*"BRSTW"*	Bairstow Method	Cntl word in Z, guesses in Y,X	shows results	*JM Baillard*
35	CHRPOL	Characteristic Polynomial	Under prgm control	Characteristic Pol Coeffs	*Ángel Martin*
36	DTC	Detele Tiny Coefficients	Cntl word in X	Deletes ak < 1E-7	*JM Baillard*
37	EIGEN	Eigen Values by SOLVE	Under prgm control	Eigen Values by Solve	*Ángel Martin*
38	EV3	Eigen values 3x3	Matrix in XMEM	Eigen Values by Formula	*Ángel Martin*
39	EV3X3	Eigen values 3x3	Prompts Matrix Elements	Eigen Values by Formula	*Ángel Martin*
40	JACOBI	Symmetrical Eigenvalues	Under prgm control	Eigen Values by Jacobi	*Valentín Albillo*
41	OPFIT	Orthogonal polynomial Fit	Under prgm control	shows results	*Eugenio Úbeda*
42	*"P+P"*	Polynomial Addition	Driver for PSUM w/CF 01	shows results	*Ángel Martin*
43	*"P-P"*	Polynomial Substraction	Driver for PSUM w/SF 01	shows results	*Ángel Martin*
44	*"P*P"*	Polynomial Multiplication	Driver for PPRD	shows results	*Ángel Martin*

#	Function	Description	Input	Output	Author
45	***"P/P"***	Polynomial Division	Driver for PDIV	shows results	*Ángel Martin*
46	**PCPY**	Copy of Polynomial	from, to cntl words in Y,X	polynomial copied	*JM Baillard*
47	**PDIV**	Euyclidean Division	cont words in Y and X	cntl words remainder & result	*JM Baillard*
48	**PEDIT**	Polynomial Editor	cntl word in X	prompts for each coeff value	*Ángel Martin*
49	**PFE**	Partial Fraction Expansion	Under prgm control	see description to decode	*JM Baillard*
50	***"PF>X"***	Prime Factors to Number	Matrix w/ Prime Facts in XMEM	restores the original argument	*Ángel Martin*
51	**PMTP**	Prompts for Polynomial	cntl word in X	prompts for complete list	*Ángel Martin*
52	**POLFIT**	Polynomial Fit	Under prgm control	calculates polynomial fit	*Valentín Albillo*
53	**POLINT**	Aitken Interpolation	Under prgm control	interpolation made	*Ulrich Deiters*
54	**POLZER**	From Poles to Zeros	Under prgm control	shows coeffs and roots	*Ángel Martin*
55	**PPRD**	Polynomial Product	cntl words in Z, Y, bbb in X	cntl word result in X	*JM Baillard*
56	***"PRMF"***	Prime Factors Decomposition	number in X	prime factors in XMEM Matrix	*Ángel Martin*
57	***"PROOT"***	Polynomial Roots	Under prgm control	Shows all roots	*Ángel Martin*
58	**PSUM**	Polynomial Sum	cntl words in Z, Y; bbb in X	cntl word result in X	*JM Baillard*
59	**PVAL**	Polynomial Evaluation	Cntl word in Y, x in X	Result in X	*JM Baillard*
60	**PVIEW**	Polinomial View	Cntl word in X	Sequential listing of coeffs	*Ángel Martin*
61	**QUART**	Quartic Equation Roots	coeefs in Stack (a4=1)	shows results	*JM Baillard*
62	***"RTSN"***	Roots subroutine	Under prgm control	calculates roots	*Ángel Martin*
63	**TOTNT**	Euler's Totient Function	argument in X	Result in X	*Ángel Martin*
64	***"#EV"***	Subroutine for EIGEN	Under prgm control	Under prgm control	*Ángel Martin*

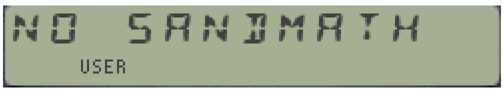
Functions in **blue** are all in MCODE. Functions in black are MCODE entries to FOCAL programs. Light blue background denotes new or improved in this revision.

I have adapted most of the FOCAL programs for optimal fit in the SandMatrix, but as you can see the original authors are always credited – including W&W for the array functions set, renamed here as '**-CCD MATRIX**". Many of the routines in this manual include the program listing, this provides an opportunity to see how the functions are used and of course adds completion to the documentation.

The function groups are distributed in both lower and upper pages, as follows:

- The lower page contains the general intro section plus the CCD Matrix set. Very much like the lower page of Advantage Pac minus the digital functions.
- The upper page has the Advanced Matrix and Polynomial sections. Basically all new and additional to the Advantage Pac.
- The second bank in the upper page is practically identical to that in the Advantage, with a few changes made after removing the Digital functions as well. It mostly contains the MCODE for the CCD Matrix functions and the major matrix calculations (**MSYS**, **MINV**, **MDET**, **TRNPS**).

The SandMath checks for the presence of its two dependencies, ie. The Library#4 and the SandMath. Note that if the SandMath module is not plugged in the calculator the following warning message is shown every time the calculator is switched on, (but not halting the polling points process):



Note: Make sure that revision "H" (or higher) of the Library#4 module is installed.

2. Lower-Page Functions in detail

The first section groups the auxiliary functions used for ALPHA string management, plus some leftover functions that either didn't belong to the other categories or were added at latest stages of the development.

2.1. SANDMATRX 4

2.1. Alpha String Management

The use of the ALPHA register for Input/Output certainly isn't new in the 41 platform, but the utilization by the Matrix functions effectively turned it into an abstraction layer for programming; therefore the importance of auxiliary utilities like these.

Some of these functions are also included in the AMC_OSX Module – yet it appeared convenient not to add it as another dependency (even if it's just a 4k footprint for its 3 banks), so here they are as well.

#	Function	Description	Input
1	**ABSP**	Alpha Back Space	Text in Alpha
2	**AIP**	Appends integer part	x in X
3	**ASWAP**	Alpha Swap	A,B in Alpha
4	**CLAC**	CLA from Comma	Text in Alpha
5	**EQT**	Displays Curve Equation	Eq# in R00 (1 – 16)
6	**ST<>A**	Exchanges Alpha and Stack	Values in Stack and Alpha registers

ABSP deletes the rightmost character in ALPHA – equivalent to "back space" in manual mode.

AIP was HP's answer to the need to append just the integer part of the number in X to Alpha – not changing the FIX and radix settings. Note also that **AIP** appends the absolute value of the number, which is not the case with **ARCLI** or **AINT** from the CCD and AMC_OS/X modules.

ASWAP handles comma-separated strings, exchanging the strings placed left and right of the *first* comma found in Alpha. Very handy to manage all those operations that have an input and output matrix names defined in ALPHA, separated by comma.

CLAC deletes the contents of ALPHA located to the right of a comma (i.e. after the comma but not including it). It is adapted from **CLA-** in the CCD Module.

EQT is an extension to the Curve Fitting functions in the SandMath. Use it to display (and write in Alpha) one of the 16 the equations available for CURVE. It requires the equation number (1 to 16) in R00. Easy does it!

ST<>A simply exchanges the contents of the stack and the four Alpha registers {M,N,O,P}. Used in 3D-vector operations where one of the operands is stored in Alpha.

2.2. Other functions in the Header section.

#	Function	Description	Input
1	**"MATRX"**	"Easy Matrix" Program	Driver for Major Matrix Ops.
2	**SQR?**	Tests for Square Matrix	Mname in Alpha
3	**MPOL**	Matrix polynomial	Mname in Alpha, Cnt'l word in X
4	**DOTN**	N-dimensional Dot product	cnt'l words in Y,X
5	**V*V**	3-dimensional Dot product	prompts for coeffs
6	**"3DV"**	3D Vectors	Prompts "|V| V* VX"

MATRX is the main driver program provided in the Advantage Pac for the major matrix calculations (MDET, MINV, SIMEQ, TRNPS). Nice and easy, maybe the only one to use for users not needing any further functionality. **MTR** was part of the same program, but has been eliminated in this revision.

The following extract describing the use of MATRX is taken from the Advantage Pac manual – and it's included here for convenience and completeness. It's useful to revise the underlying concepts as well.

2.2.1 The Matrix Program

This chapter describes the matrix program, **MATRX** - the easy, "user-friendly" way to use the most common matrix operations on a newly created matrix. To use **MATRX** you do not need to know how the calculator stores and treats matrices in its memory. The next chapter lists and defines every matrix function in the pac, including those called by **MATRX.** Using these functions on their own requires a more intimate knowledge of how and where the calculator stores matrices.

What this program can do.

Consider the equations:

$$3.8\ x_1 + 7.2\ x_2 = 16\ .5$$
$$1.3\ x_1 - 0.9\ x_2 = -22.1$$

for which you must determine the values of x1 and x2 . These equations can be expressed in matrix form as **AX** = **B**, where **A** is the coefficient matrix for the system, **B** is the column or constant matrix, and X is the solution or result matrix.

$$\mathbf{A} = \begin{bmatrix} 3.8 & 7.2 \\ 1.3 & -0.9 \end{bmatrix}, \quad \mathbf{X} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}, \quad \text{and } \mathbf{B} = \begin{bmatrix} 16.5 \\ -22.1 \end{bmatrix}$$

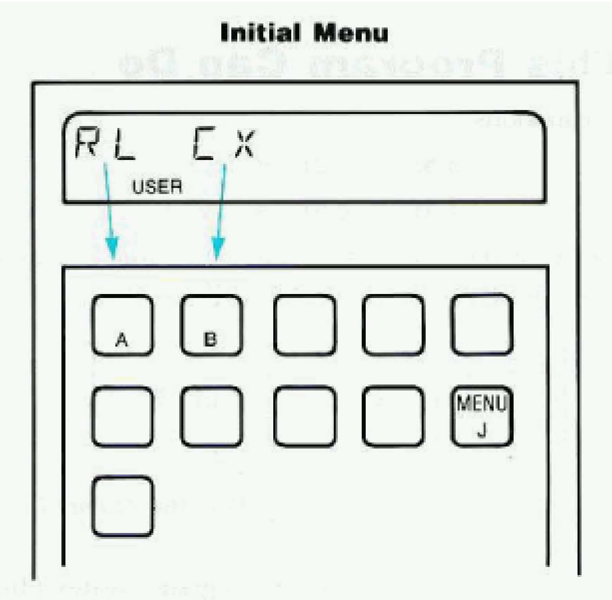
For such a matrix system, the **MATRX** program creates (dimensions) a square real ar complex matrix, **A**, and a column matrix, **B.** You can then:

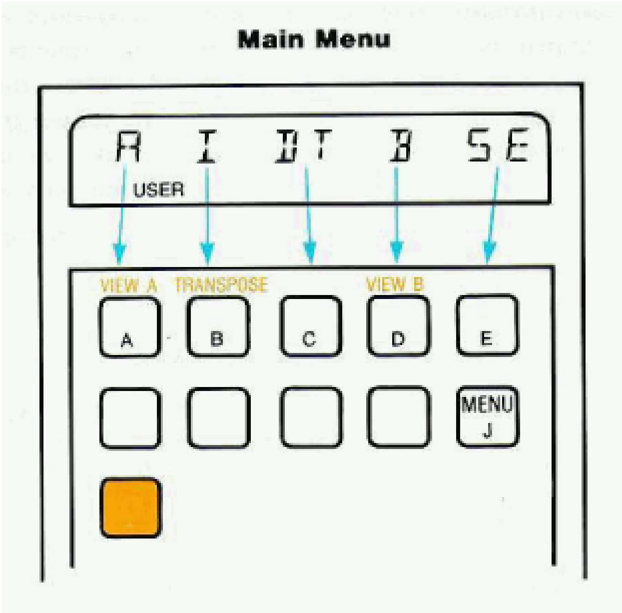
- Enter, change ('edit"), or just view elements in **A** and **B.**
- Invert **A.**
- Transpose **A** if **A** is real.
- Find the determinant of **A** if **A** is real.
- Solve the system of simultaneous equations by finding the solution to **AX = B.**

The size of your matrix is limited only by available memory (each real matrix requires one register pIus one register for each element.) If you want to store more than one matrix, you will need to use the matrix function **MATDIM**, described in the next chapter. The **MATRX** program does not store or recall matrices; it works with a single square matrix A and a single coIumn matrix B. When you enter new elements into A you destroy its old elements.

Instructions

MATRX has two menus to show you which key corresponds to which function. The initial menu you see is to select a real ox complex matrix: (picture on the left below)





After you make this selection, input the order of the matrix, and press R/S, you will see the main menu (picture on the right above). This menu shows you the choice of matrix operations you have in **MATRX**. Press [**J**] to recall this menu to the display at any time. This will not disturb the program in any way.

To clear the menu at any time press "Back Arrow". This shows you the contents of the X-register, but does not end the program. You can perform calculations, and then recall the menu by pressing [**J**]. (However you don't need to clear the program's display before performing calculations.)

- The program starts by asking you for a new matrix. It has you specify real vs. complex and the order (dimension) of a square matrix for A.
- The program does not clear previous matrix data, so previous data – possible meaningless data – will fill your new matrices A and B until you enter new values for their elements.
- Each element of a complex matrix has two values (a real part and an imaginary part) and requires four times as much memory to store as an element in a real matrix. The promprs for real parts x11, x12, etc. are "**1:1=**?", "**1:2=?**", etc. The prompts for complex parts x11+ i y11, x2+ i y22, etc. are "**RE.1:1=?**", "**IM.1:1=?**", etc.

Remarks

Alteration of the Original Matrix. The input matrix A is altered by the operations finding the inverse, the determinant, the transpose and the solution of the matrix equation. You can re-invert $\mathbf{A}^{-1}$, and re-transpose $\mathbf{A}^{T}$ to restore the orignal form of **A**. However, if you have calculated the determinant or the solution matrix, then **A** is in its LU-decomposed form. To restore **A**, simply *invert it twice*. The LU-decompostion does not interfere with any subsequent **MATRX** operation except transposition and editing (do not attempt to edit an LU-decomposed matrix unless you intend to change *every* element). For more information on LU-decomposition, refer to "LU-Decomposition" in the next chapter ('Matrix Functions").

Matrix Storage. The **MATRX** program stores a matrix **A** starting in R0 of main memory; it is named "**R0**". Its column matrix **B** is stored after it, and the result matrix **X** overwrites **B**, Refer to the chapter "Matrix Punctions' for an explaniation of how matrices are named and stored, and how much room they need. **MATRX** cannot access any other matrices, with the exception of the previous **R0** and its corresponding column matrix.

Redefined Keys. This program uses local Alpha labels (as explained in the owner's manuaI for the HP-41) assigned to keys [A]-[E], [J] , [a], [b], and [d]. These local assgnments are overriden by any User-key assigments you might have made to these same keys, thereby defeating this program. Therefore be sure to clear any existing User-key assignments of these kep before using this program, and avoid redefining these keys in the future.

Example 1.

Given the system of equations at the beginning of this section, find the inverse, determinant and transpose of A, and then find the solution matrix of the equation AX = B

$$\begin{bmatrix} 3.8 & 7.2 \\ 1.3 & -0.9 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 16.5 \\ -22.1 \end{bmatrix}$$

Keystrokes	Display	Comments
XEQ "**MTRX**"	"RL CX"	Starts the MTRX program
[**A**] (RL)	"ORDER=?"	Selects a real Matrix
2, R/S	"A I DT B SE"	Dimensions a 2x2 square matrix
[**A**]	"1:1=a11?"	Enters the Editor and displays old value
3.8, R/S	"1:2=a12?"	enters the new value for a_{11}
7.2, R/S	"2:1=a21?"	
1.3, R/S	"2:2=a22?"	
.9, CHS, R/S	"A I DT B SE"	enters a_{22} and returns main menu
[**B**] (I)	"A I DT B SE"	Inverts A
[SHIFT][**A**]	"1:1=0.0704"	Displays the current contents
R/S	"1:2=0.5634"	of A after the inversion
R/S	"2:1=0.1017"	
R/S	"2:2=-0.2973"	
R/S	"A I DT B SE"	
[**B**] (I)	"A I DT B SE"	Re-inverts A^{-1} to the original
[SHIFT][**B**]	"A I DT B SE"	Transposes A
[SHIFT][**A**]	"1:1=3.8000"	Displays the current contents
R/S	"1:2=1.3000"	of A after the transposition
R/S	"2:1=7.2000"	
R/S	"2:2==0.9000"	
R/S	"A I DT B SE"	
[SHIFT][**B**]	"A I DT B SE"	Re-transposes A^T to the original A
[**C**] (DT)	"DET=-12.7800"	Det(A)
[**B**]	"1:1=b11?"	Enters the editor for B and displays old elements
16.5, R/S	"2:1=b12?"	Enters the new value for b_{11}
22.1, CHS, R/S	"A I DT B SE"	Enters b_{22} and returns main menu
[**E**] (SE)	"A I DT B SE"	Solves the system AX = B, placing X in B
[SHIFT] [**D**]	"1:1=-11.2887"	displays the solution matrix
R/S	"2:1=8.2496"	
R/S (or [**J**])	"A I DT B SE"	Exits the editor

Example 2. Find the inverse of the complex matrix:

$$\begin{bmatrix} 1+2i & 3+3i \\ 4+5i & 6+7i \end{bmatrix}$$

Note that the original **MATRX** has been slightly edited in the SandMatrix so that the program sets the required SIZE if not enough registers are currently available to store the matrices – so you don't need to worry about that mundane detail. This example is also interesting because also shows how to make corrections to the data entered by mistake.

Keystrokes	Display	Comments
XEQ "**MATRX**"	"RL CX"	Starts the MTRX program
[**B**] (CX)	"ORDER=?"	Selects a complex Matrix
2, R/S	"A I DT B SE"	Dimensions a 2x2 complex matrix
[**A**], R/S	"RE1:1=x11?"	Enters the editor and sisplays old value
1, R/S	"IM1:1=y11?"	ditto for the imaginary part
2, R/S	"RE1:2=x12?"	
3, R/S	"IM1:2=y12?"	
4, R/S	"RE:2:1=x21?"	Wrong entry! Should be 3, not 4...
1,002, [**A**]	"RE1:2=3.000?"	Moves editor back to x_{12}
R/S	"IM1:2=4.000?	The wromg imaginary part
3, R/S	"RE2:1=x21?"	Correct value is entered for y_{12}. Proceed
4, R/S	"IM2:1=y21?"	
5, R/S	"RE2:2=x22?"	
6, R/S	"IM2:2=y22?"	
7, R/S	"A I DT B SE"	Enters last element and returns main menu
[**B**] (I)	"A I DT B SE"	Inverts A
[SHIFT][**A**]	"RE1:1=-0.9663"	Viewing A-1
2.002, [**A**]	"RE2:2=-0.2369"	Displays x_{22} + i y_{22}
R/S	"IM2:2=-0.0225"	
R/S (or [**J**])	"A I DT B SE"	Exits the editor

Other (more advanced) examples are available in the next sections of the manual, during the description of the individual matrix functions.

2.2.2.- Matrix Polynomial (MPOL)

MPOL was a last-minute addition to the ROM, which somehow combines both matrix and polynomial algebra. Use it to calculate a matrix polynomial P(A) - not to be confused with a polynomial matrix - based on an existing **square** matrix [A] and a polynomial P(x).

P(A) is the result matrix calculated replacing the real variable x with [A], using the polynomial coefficients to multiply the different matrix powers as per the order of the polynomial terms. As it's the case all throughout polynomials, Honer's method proves very useful to reduce all the matrix powers to matrix multiplications – with considerable execution time reduction and simplification of the code.

Example.- Calculate P{A) for the following matrix and polynomial:

$P(x) = 2x^4 - x^3 + 3x^2 - 4x + 5$; and:

```
     [[ 4  2  3 ]
A  =  [ 3  2  5 ]
      [ 2  1  4 ]]
```

This is also a good example to become familiar with the editor and input routines available in the SandMatrix. First we'll create and populate the matrix using the **Matrix Editor** input functionality – very recommended for integer elements, as follows:

ALPHA, "A", ALPHA, 3,003, XEQ "**MATDIM**" creates the matrix in X-Mem, then:

XEQ "**PMTM**" -> at the prompt "R1: _" we type: 4, ENTER^, 2, ENTER^, 3, R/S
-> at the prompt "R2: _" we type: 3, ENTER^, 2, ENTER^, 5, R/S
-> at the prompt "R3: _" we type: 2, ENTER^, 1, ENTER^, 4, R/S

The Matrix has been completely input using "batches" (or lists) including all elements of each row simultaneously – this is an advantageous way to handle them that results in faster and less error-prone method, not based on a single-element prompt.

Note how pressing ENTER^ during this process results into a blank space in the display separating each of the elements, and that the sequence is terminated pressing R/S. Upon completion the matrix elements are stored in the Matrix file in extended memory.

The analogous function for the polynomial is **PMTP**, which requires the control word in x – a number of the form **bbb.eee**, denoting the beginning and ending registers that contain the polynomial coefficients. In this case:

2.006, XEQ "**PMTP**" -> at the prompt "R2: _" we type:
2, ENTER^, CHS, 1, ENTER^, 3, ENTER^, CHS, 4, ENTER^, 5, R/S

Note how in this case the function knows there's no more "rows" to add, and also that negative values are easily input using the CHS key. Upon completion the coefficients are stored in registers R01 to R05.

The last step is executing **MPOL** itself. To do that we place the matrix name in Alpha and the polynomial control word in X, then call **MPOL**. The resulting P(A) is stored in a new matrix named "**P**" - also located in an XM file - therefore [A] is not overwritten. Note however that this will overwrite [P] if it already exists. In this case we have:

```
        [[ 3548  1887  4705 ]
P(A) =   [ 3727  1987  4962 ]
         [ 2539  1351  3385 ]]
```

The result matrix name is placed in ALPHA when the execution ends, so you can directly use any matrix editor routine (like **OMR**) to review its elements. Note how **OMR** *will display integer values without any zeros after the decimal point, regardless of the current FIX settings*. Set flag 21 to stop the display of each individual element.

In addition to the result matrix P(A), **MPOL** also requires an auxiliary matrix for intermediate calculations. The matrix file "#" is temporarily created during the execution for this purpose, and deleted upon completion of the program. While this is transparent to the user you may want to remember this fact due to the extended memory needed to allow for it – with a total of 3 x (n^2 + 2) registers used (including the file headers).

The last point to remember about **MPOL** is that it uses data registers R00 and R01 – which therefore *cannot be used to store the polynomial coefficients*.

- R00 has the polynomial control word and is used as counter for the loop execution
- R01 has the matrix name. It's left unchanged.

Below you can see the program listing for **MPOL** – not a long program, albeit not as short as a simple polynomial evaluation for real variables. Note the use of function **I#J?** to check for square matrix, as well as the "shortcut" **-ADV MTRX** that puts the error message "*NOT SQUARE*" in the display and terminates the execution.

01	LBL "MPOL"		23	"P,"	
02	DIM?		24	ARCL 01	
03	I#J?	*is it square?*	25	"|-,#"	"P,A,#"
04	-ADV MTRX	*no, prompt error*	26	M*M	
05	RDN	*cnt'l word to X*	27	"#,P"	
06	E-3		28	CLST	
07	-		29	MMOVE	
08	STO 00		30	ISG 00	*next index*
09	ASTO 01		31	GTO 00	*loop back*
10	DIM?		32	XEQ 02	
11	"P"		33	PURFL	*purge auxiliary mat*
12	MATDIM		34	MNAME?	*bring result name*
13	"#"		35	RTN	
14	MATDIM		36	LBL 02	
15	"X,"		37	"#"	
16	ARCL 01		38	MIDN	
17	",P"	"X,A,P"	39	"X,#,#"	
18	RCL IND 00		40	RCL IND 00	*next coeff*
19	MAT*	*initial value*	41	MAT*	
20	ISG 00	*next index*	42	"#,P,P"	
21	LBL 00		43	MAT+	*add it to partial result*
22	XEQ 02		44	END	

The auxiliary matrix "#" is needed because unfortunately **M*M** does not allow the result product matrix to be the same as any of the multiplication factors. At least we double-use it for other intermediate calculations as well (identity matrix products), killing two birds with the same stone.

MPOL is representative of the kind of routine that makes the extensions to the base matrix functions set of the Advantage – hopefully it has whet your appetite and are looking forward to seeing more… and that we will in later sections of the manual.

2.2.3.- N-dimensional Vector Operations

DOTN is an all-MCODE implementation of a n-dimensional vector dot (scalar) product, the norms of each operand and the angle between them. Originally written by JM Baillard, the input parameters are the control words for each vector in registers X and Y (more about this later), and the result value are placed in the stack.

Obviously the vector components must be input in the appropriate registers, which you can do using any of the available input programs available in the SandMatrix – will be seen with detail in the polynomial section later in the manual. Incidentally the code for **DOTN** is located in the second bank of the upper page – taking advantage of the available room after the removal of the digital functions.

Example. Calculate the scalar product of vectors U(2,3,7,1) and V(3,1,4,6), storing their components in registers {R01 - R04} for U, and {R06 - R09} for V.

For the data input we have several choices; here we'll Use the **PMTP** function seen before, just pretending the vector components are analogous to polynomial coefficients (which is irrelevant to the actual workings of **PMTP**).

1.004, XEQ "**PMTP**" -> "R1: _", we type: 2, ENTER^, 3, ENTER^, 7, ENTER^, 1, R/S
6.009, XEQ "**PMTP**" -> "R6: _", we type: 3, ENTER^, 1, ENTER^, 4, ENTER^, 6, R/S

Re-entering the control codes in X, and Y we execute the function, which returns:

XEQ "**DOTN**" -> 43,, see table below for all the available data.

STACK	INPUTS	OUTPUTS	Results
T	/	**µ**	46.52626239°
Z	/	**|| U ||**	7.874007874
Y	bbb.eee(U)	**|| V ||**	7.937253933
X	bbb.eee(V)	**U.V**	43,000000
L	/	cos µ	

A good example of Jean-Marc's very complete and economical programming. Needless to say it executes at blazing light speed, as you would expect from an MCODE routine like this.

The alternative – Vectors as Matrices.

V*V performs the same tasks (n-dimensional vector dot product) but using a different approach: treating the vectors as column matrices it simply uses **M*M** to calculate the resuilt, multiplying the first operand vector by the transpose of the second operand vector. All data input/output are driven under program contol. The execution time is longer than **DOTN**, trading so convenience for speed.

To appreciate the workings of **V*V** you need to consider that it transposes V2 before doing the multiplication, and that it calculates the Frobenius norms of each matrix on the fly to obtain the angle. The dot product is placed in a 1x1 matrix named "V*V" in X-Mem.

Here's the listing of the program that clearly shows all the housekeeping chores required to prepare the strings needed in ALPHA for the matrix functions as input. Even if it's somehow slower and less efficient, it's a good "academic example" of utilization of the standard matrix functions.

01	LBL "V*V"		31	FNRM	
02	FS? 06	*subroutine use?*	32	/	
03	*GTO 00*	*yes, skip data entry*	33	*"V2"*	
04	-SNDMTRX 4	*prompts "ORDER=?"*	34	FNRM	
05	STOP		35	/	
06	INT		36	ACOS	
07	*"V1"*		37	X<>Y	
08	MATDIM		38	*"V<)="*	
09	XEQ 05	*V1 data input*	39	ARCL Y	
10	DIM?		40	PROMPT	*show angle*
11	*"V2"*		41	RTN	
12	MATDIM		42	LBL 05	
13	XEQ 05	*V2 data input*	43	3	
14	LBL 00		44	X<>F	
15	*"V*V"*		45	0	
16	CLX		46	MSIJA	*position pointer*
17	MATDIM		47	LBL 04	
18	*"V1"*		48	*"c"*	
19	TRNPS		49	MRIJ	
20	*"|-,V2,V*V"*		50	MP	
21	M*M		51	MR	
22	ASHF		52	ARCLX	
23	0		53	*"|-?"*	
24	MSIJA	*position pointer*	54	PROMPT	
25	MR	*recall element*	55	MS	
26	ENTER^		56	I+	
27	*"|-="*		57	FC? 10	*reached the end?*
28	ARCL X		58	*GTO 04*	*no, loop back*
29	PROMPT	*show result*	59	MNAME?	
30	*"V1"*		60	END	

The usage of user flag 06 determines whether the program is used as a subroutine – in which case the data entry is skipped. This is more or less consistently done throughout the SandMatrix module, and has the benefit of saving one entry in the FAT – which would be needed for the subroutine label.

Line 4 uses the header function "-SNDMTRX 4", which in program mode adds the text "ORDER=?" to the display (not ALPHA). This saves bytes and keeps the contents of ALPHA unchanged.

2.2.4.- 3D Vectors Mini-Calculator.

Lastly **"3DV"** is a mini-vector calculator; use it to calculate the Module of a vector, or the DOT and CROSS products of two 3D vectors. It's basically a small menu-driven shell that uses functions **VMOD**, **V*A**, and **VXA** available in the auxiliary FAT within the SandMath. One of the operand vectors is placed in ALPHA registers {M,N,O}, therefore their names.

Its prompt looks like this:

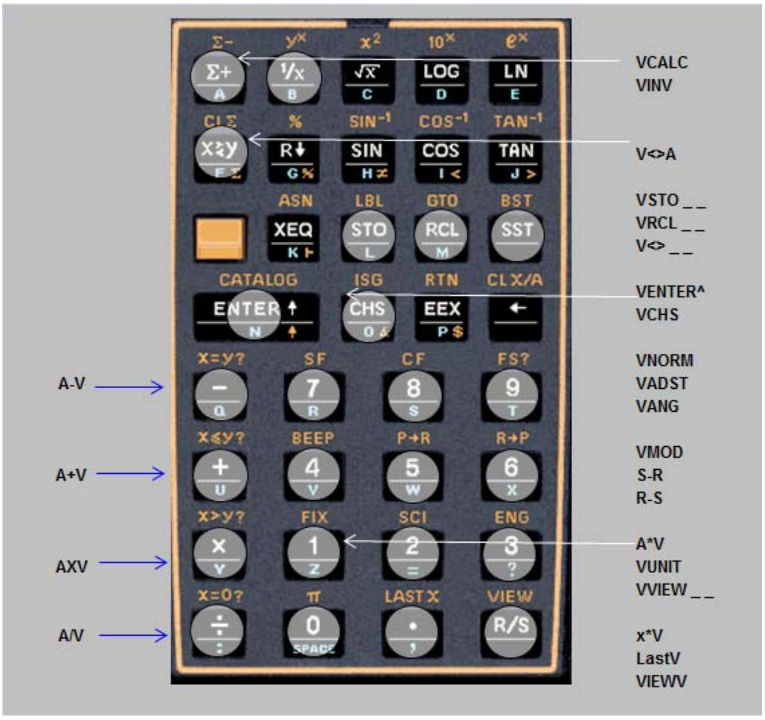
Which assumes no assignments are done on the [**A**], [**C**], and [**E**] keys and that USER mode is on.

Data entry is also under program contol, and nice alphanumeric mnemonics describe the result(s). The module and the dot product are left in X upon completion. For the cross product case the three components are sequentially displayed, with a pause in between them. They're also placed in the stack registers Z,Y,X for subroutine use.

The program listing is below – note how this trivial little application manages to make good use of some of the sub-functions in the SandMath module, as well as the interesting way to use the ALPHA register for the vector components.

1	**LBL "3DV"**		25	**ARCL X**	
2	**LBL 02**	*new start*	26	**AVIEW**	
3	**CF 00**		27	**PSE**	
4	*"|V1| Vx VX"*	*menu options*	28	*"VY=*	
5	**SF 27**	*User mode ON*	29	**ARCL Y**	
6	**PROMPT**		30	**AVIEW**	
7	**LBL A**	*Modulus / Norm*	31	**PSE**	
8	**SF 00**		32	*"VZ="*	
9	**XEQ 05**		33	**ARCL Z**	
10	**VMOD**	*SandMath's*	34	**AVIEW**	
11	*"|V|=*		35	**PSE**	
12	**GTO 00**		36	**GTO 02**	*start over*
13	**LBL C**	*DOT product*	37	**LBL 05**	
14	**XEQ 03**		38	*"^V1=?"*	*prompt for V1*
15	**V*A**	*SandMath's*	39	**PROMPT**	
16	*"V*="*		40	**FS? 00**	*module?*
17	**LBL 00**		41	**RTN**	*yes, go back*
18	**ARCL X**		42	*"^ V2=?"*	*prompt for V2*
19	**PROMPT**		43	**CF 21**	
20	**GTO 02**	*start over*	44	**AVIEW**	*display first,*
21	**LBL E**	*CROSS product*	45	**ST<>A**	*then exchange*
22	**XEQ 05**		46	**STOP**	
23	**VXA**	*SandMath's*	47	**END**	
24	*"VX="*				

You're encouraged to check the **Vector Analysis ROM** for a comprehensive implementation of a 3D-Vector calculator, as well as other geometry programs. The Vectors ROM is completely self-contained, and only takes up one page (4k), complementing the SandMatrix (and the SandMath) very effectively.



2.2. MATRIX 101

2.2.1. Setting up a matrix: Name, Storage, and Dimension

The first group of matrix functions are used to create, populate and store the matrices.

	Function	Description	Inputs
1	**MATDIM**	Dimensions a Matrix	Name in Alpha, dimensions in X
2	**MNAME?**	Returns name of current Matrix to Alpha	none
3	**DIM?**	Returns the dimension of Matrix	Name in Alpha
4	**"MEDIT"**	Matrix Editor	Name in Alpha
5	**"CMEDIT"**	Complex Matrix Editor	Name in Alpha

You can create, manipulate, and store real and complex matrices. The size and number of matrices is limited only by the amount of memory available in the calculator. If you have extended memory you can also store matrices there.

To create a matrix you must provide its name and dimensions. The function **MATDIM** uses the text in the Alpha register as its name, and the dimensions mmm.nnn in the X-register to create a matrix. It does not clear (zero) the elements of a new matrix in main memory, but retains the existing contents of the previous matrix or registers. *It does clear the elements of a new matrix in extended memory*. You then enter values- numeric or Alpha- into a matrix via the matrix editors.

Naming a Matrix

The name you give a matrix determines where it will be stored. A matrix to be stored in main (non-extended) memory must be named **Rxxx**, where xxx is up to three digits. (You can drop leading zeros.) The matrix will be stored starting in Rxx. For example, **R007** is the same as **R7**, which would store this matrix header in R07. As a shortcut, if you specify matrix **R**, its name and location will be R0.

A matrix to be stored in extended memory can be named with up to seven Alpha characters, excepting just the letter "X" (which is reserved to name the X-register) and the letter "R" followed by up to three digits (which is reserved to name the main memory arrays). You do not need to specify a file type; it will automatically be given one unique to matrices. Use the Alpha register to specify matrix names. When specifying more than one name (as parameters for certain functions), separate them with commas.

Dimensioning a Matrix

Specify the dimensions of a new matrix as mmm.nnn, where m is the number of rows and n is the number of columns. You can drop leading zeros for m and trailing zeros for n. For a complex matrix, specify mmm.nnn as *twice* the number of rows and *twice* the number of columns. (Refer to "Working with Complex Matrices"). A zero part defaults to a 1, so 0 is equivalent to 1.001, 3 to 3.00 1, and .023 to 1.023.

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \qquad \begin{bmatrix} 1+i & 2+3i \\ 4+5i & 6+7i \end{bmatrix}$$

$$mmm.nnn = 2.003 \qquad mmm.nnn = 4.004$$

- **MATDIM** Dimensions a new matrix or redimensions an existing one to the given dimensions.
- **MNAME?** Returns the name of the current matrix to the Alpha register.
- **DIM?** Returns the dimensions mmm.nnn of the matrix specified in the Alpha register to the X-register. (A blank Alpha register specifies the current matrix.)

How a Matrix Is Stored

The elements of a matrix are stored in memory in order from left to right along each row, from the first row to the last. Each element occupies one data-storage register. A complex number requires four registers to store its parts.

Memory Space.- A matrix in main memory occupies (m x n) + 1 datastorage regis ters, one register being used as a status header. A complex matrix uses (2m x 2n) + 1 registers, where m is the number of rows in the complex matrix and n is the number of columns in the complex matrix.

A matrix in extended memory has a file length of m x n . (2m x 2n for a complex matrix). Its file type is unique to matrices. Do not use the function **CLFL** with a matrix in extended memory: this destroys part of the file's header information. Instead, use **PURFL** to purge the entire matrix.

Changing Matrix Dimensions.- If you redimension a matrix to a different size, then the existing elements are reassigned to new elements according to the new dimensions. Extra old elements are lost; extra new elements take on the values already present in the new registers- except in extended memory, where new elements are set to zero.

Redimensioning 2 x 3 to 2 x 2 :

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \begin{array}{c} \text{lost} \\ 5 \quad 6 \end{array}$$

Redimensioning 2 x 3 to 2 x 4 :

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & ? & ? \end{bmatrix}$$

This is what happens each time you dimension a new matrix since the old elements from the previous current matrix remain until you change them.

Caution.- When **MATDIM** is used to redimension a matrix stored in extended memory, the position of the matrix pointer is not readjusted. If the pointer happened to be positioned to an element that is outside the new bunds of the redimensioned matrix, it must be repositioned to be within the new bounds by executing either MSIJ or MSIJA with valid indices before the pointer can be used again.

Existing matrices in extended memory cannot be redimensioned to completely fill extended memory. The maximum allowable size of a redimensioned matrix is one register less than the currently available extended memory. A new matrix can, however, be dimensioned to completely fill available extended memory.

Using the Matrix Editors

There are two matrix editors: **MEDIT** for real matrices and **CMEDIT** for complex matrices. They are otherwise quite similar. The matrix editors are used for three purposes:

- Entering new values into the elements of a matrix.
- Reviewing and changing (editing) the elements of a matrix, either in order or by "random access" of specific elements.
- Viewing (without being able to change) the elements of a matrix (flag 08 set).

When you execute MEDIT or CMEDIT, the editor displays element 1,1 of the matrix specified in the Alpha register or of the current matrix if the Alpha register is empty. Pressing R/S steps the display through the elements; for a complex matrix, each part of the complex element is shown separately.

Function	Display	Function	Display
MEDIT	1:1=1.0000?	CMEDIT	RE.1:1=1.0000?
R/S	1:2=2.0000?	R/S	IM.1:1=1.0000?
		R/S	RE.1:2=2.0000?
⋮	⋮	⋮	⋮
R/S	*(X-register)*	R/S	*(X-register)*

The **"?"** at the end of the display line indicates that you can change that value. In effect, you are being asked whether this is the value you want. If you want to change the element you see, just enter the new value and press R/S. You do this for a brand new matrix as well as for correcting or altering a single value. If you press R/S without entering a new value, the current value remains unchanged.

Viewing without editing.- If you set flag 08, the editor will let you only view the elements, not change them. The display appears without the "**?**" at the end of the line. 1:1= 1.0000
If you have a printer attached while flag 08 is set, it will print out all the elements of the matrix without pausing.

Directly accessing any element.- You can directly access any specific element while the editor is active (and the User keyboard is also active). To access the element in the i-th row and the j-th column, enter iii.jjj and press [**A**]. (This is as in the MATRX program.) You can drop leading zeros in iii and trailing zeros in jjj. For a complex matrix, you can directly access the real pari of element i, j .Then use R/S to access its imaginary part. You can drop leading zeros in the i-part and trailing zeros in the j-part. *A zero part defaults to a 1*.

Exiting the Editor.- To leave the editor before it has reached the last element, do either:
- Press [**J**].
- Try to access a nonexistent element. For instance, in a 4 x 4 matrix, press 5 [**A**].

How to Specify a Matrix

Given the matrix multiplication operation **AB = C**, you know **A** and **B** and are looking for the product matrix, **C**. In performing this operat ion, the calcula tor must be given the identit ies of the existing matrices **A** and **B**, and also be told where to put the result matrix, **C**. (However, the result matrix can be the same as one of the input matrices.) All given matrices must al ready exist as named, dimensioned matrices. Naturally, only **A** and **B** must contain valid data.

Some functions use only one input matrix, and some functions automatically use one of the input matrices for output. So the minimum number of matrices to specify is one, and the maximum is three.

A matrix function checks the Alpha register for the names (that is, the locations) of the matrices it needs for input and output. Before executing that function, you should specify all needed parameters on one line in the Alpha register, separating each with a comma:

Alpha Register	*input matrix[,input matrix][,result matrix]*

Scalar Operations.- Scalar input and output must be in the X-register, and so this location does not need to be specified unless the function in question can use *either* a scalar *or* a matrix for the same input parameter. To specify the X-register, use **X**.

For instance, **MATDIM** requires a scalar input and a matrix name, so you do not need to specify the X- register. On the other hand, the scalar arithmentic functions, such as **MAT+**, can use either two matrices or a scalar and a matrix for input. Therefore, you must specify **X** if you want to use it.

The Current Matrix.- The current matrix is the last one accessed (used) by a matrix operation. If the Alpha register is clear and you execute a matrix function that requires a matrix specification, the current matrix is used by default. (If there is no current matrix, "UNDEF ARRAY" results).

The result matrix of a matrix function becomes the current matrix following that operation. To find out the name of the current matrix, execute **MNAME?**. Its name is returned into the Alpha register, *overwriting* its previous contents.

Default Matrix Parameters.- If you don't specify any or all the matrices that a matrix function needs, then certain default parameters exist. (Default parameters are those automatically assumed if you don't specify them.) The most common default you will probably use is the current matrix. If you don't specify a particular matrix name and the Alpha register is clear, then the default matrix is the current one.

For matrix operations requiring up to three matrix names in the Alpha register, the following table gives the conventions to interpret the parameters.

Alpha Register's Contents	Matrices Specified
A,B,C	A, B, C
A,B	A, B, B
A	A, A, A
A,,B	A, A, B
,A,B	current, A, B
,A	current, A, A
,,A	current, current, A
X,A,B	X-reg, A, B
X,A	X-reg, A, A
A,X	A, X-reg, A
A,,X	A, A, A (ignores X)
X	X-reg, current, current
(blank)	current, current, current

2.2.2.- Storing and Recalling individual Matrix elements.

The matrix editor provides a method of storing and reviewing matrix elements. For programming, you can use the following functions to manipulate individual matrix elements. A specific element is identified by the value *iii.jjj* for its location in the i-th row of the j-th column. You can drop leading zeros in the i-index and trailing zeros in the j-index. The value of the pointer defines the *current element*.

Setting and recalling the Pointer

	Function	Description	Inputs
1	**MSIJA**	Sets element pointer of matrix in Alpha	Name in Alpha, iii,jjj in X-reg.
2	**MSIJ**	Sets element pointer of current matrix	iii,jjj in X-reg.
3	**MRIJA**	Recalls element pointer of Matrix in Alpha	Name in Alpha, iii,jjj in X-reg.
4	**MRIJ**	Recalls element pointer of current matrix	iii,jjj in X-reg.

The following functions increment and decrement the element pointer rowwise (iii) or column wise (jjj). If the end of a column is reached (with the i-index) or the end of a row is reached (with the j-index), then the index advances to the next larger or smaller column or row and sets flag 09. If the index advances beyond the size of the matrix, both flags 09 and 10 are set. These functions always either set or clear flags 09 and 10. If the conditions listed above don't occur, the flags are cleared every time the functions are executed.

Incrementing and Decrementing the Pointer

The following functions were not in the original CCD ARRAY FNS group, therefore are HP's:

	Function	Description	Inputs
5	**I+**	Increments iii pointer by one	none
6	**I-**	Decrements iii pointer by one	none
7	**J+**	Increments jjj pointer by one	none
8	**J-**	Decrements jjj pointer by one	none

Storing and Recalling the Element's Value. (alone or sequentially)

The following functions provide a faster, more automated alternative to adjusting the pointer value to access each element. These combine storing or recalling values and then incrementing or decrementing the i- or j-index, so that the pointer is automatically set to the next element.

	Function	Description	Inputs
9	**MS**	Stores value in X-reg into current element	Value in X-Reg
10	**MR**	Recalls current element to X-reg	None. Returns element to X-reg
11	**MSC+**	Stores value in X-reg to current element and advances pointer to *next* element in **column**	Value in X-reg.
12	**MSR+**	Stores value in X-reg to current element and advances pointer to *next* element in **row**	Value in X-reg.
13	**MRC+**	Recalls current element to X-reg and then advances pointer to *next* element in **column**	None. Returns element value to X-reg
14	**MRR+**	Recalls current element to X-reg and then advances pointer to *next* element in **row**	None. Returns element value to X-reg
15	**MRC-**	Recalls current element to X-reg and then decrements pointer to *previous* in **column**	None. Returns element value to X-reg
16	**MRR-**	Recalls current element to X-reg and then decrements pointer to *previous* one in **row.**	None. Returns element value to X-reg

When the end of a column or row is reached, the pointer's index advances to the next (or previous) column or row. If the pointer's index is moved beyond the boundaries of the matrix, it cannot be moved back using these functions. You must use **MSIJ** or **MSIJA** .

The following sequence of keystrokes will create the matrix **ABC** (in extended memory).

$$ABC = \begin{bmatrix} 5 & 6 & 7 \\ 8 & 9 & 10 \end{bmatrix}$$

Keystrokes	Display	Comments
ALPHA, "ABC", ALPHA		
2.003, XEQ "**MATDIM**"	2.003	Dimensions matrix ABC in X-Mem.
0, XEQ "**MSIJA**"	0,000	Sets pointer to 1.001 position
5, XEQ "**MSR+**"	5.000	Enters element and advances pointer to next column for next entry
6, XEQ "**MSR+**"	6.000	Ditto as above
7, XEQ "**MSR+**"	7.000	Pointer automatically moves to second row, also setting flag 09.
8, XEQ "**MSR+**"	8.0000	
9, XEQ "**MSR+**"	9.0000	
10, XEQ "**MSR+**"	10.0000	This sets both flags 09 and 10.
SF 08		This sets the editor to display only.
XEQ "**MEDIT**"	"1:1=5.0000"	
R/S	"1:2=6.0000"	
R/S	"1:3=7.0000"	
R/S	"2:1=8.0000"	
R/S	"2:2=9.0000"	
R/S	"2:3=10.0000"	

Updated Matrix Editor: Row Input mode.

From the examples of **MPOL** we have already seen another, more effective way to enter the element values – using **PMTM** (instead of **MEDIT**) to handle them "*one row at a time".* This drastically speeds up the process, although some limitations apply:

- The maximum length for all values and the blank spaces in between them is 24 characters, as it uses the Alpha register to temporarily hold them.
- Decimal and negative values are supported in this mode, but values with exponential notation (i.e. 2.4 E23) cannot be entered using **PMTM**.

Here's the how the sequence would change using this approach:

Keystrokes	Display	Comments
ALPHA, "ABC", ALPHA		
2.003, XEQ "**MATDIM**"	2.003	Dimensions matrix ABC in X-Mem.
XEQ "**PMTM**"	"R1:"	prompts to enter the first row
5, ENTER^, 6, ENTER^, 7, R/S	"R2:"	prompts for the second row
8, ENTER^, 9, ENTER^, 10, R/S		done!

2.3. M-FUNCTIONS

This section briefly defines the matrix functions besides the dimensioning, storing, and recalling functions discussed above. Note that most of these functions are not meaningful for matrices containing Alpha data and that many of these functions are not meaningful for complex matrices. In any case. a complex matrix appears as a real matrix to all functions except **CMEDIT**. Refer to "Working with Complex Matrices'" for more information on using these functions with complex matrices.

2.3.1. Matrix Arithmetic

	Function	Description	Input
1	**MAT+**	Adds scalar or element to each element	A,B,C, or X,B,C in Alpha
2	**MAT-**	Subtracts scalar/element to each element	A,B,C, or X,B,C in Alpha
3	**MAT***	Multiplies scalar/element to each element	A,B,C, or X,B,C in Alpha
4	**MAT/**	Divides each element by scalar or element	A,B,C, or X,B,C in Alpha
5	**M*M**	Calculates the true matrix product	A,B,C in Alpha

The matrix arithmetic functions provided are scalar addition, subtraction, multiplication, and division, as well as true matrix multiplication. The scalar arithmetic functions can use two matrices as operands, or one scalar and one matrix. When using two matrices, the matrices do not have to be of the same dimension, but the total number of elements in each must be the same. This also applies to the result matrix. (Note that the i-j notation below assumes that the dimensions of the matrices are the same. If this is not the case, the i-j notation does not apply.)

Matrix multiplication, on the other hand, calculates each new element by summing the products of the first matrix's row elements by the second's column elements. The number of columns in the first matrix must equal the number of rows in the second matrix. The result matrix must have the same number of rows as the first matrix and the same number of columns as the second matrix.

If there is a scalar operand, it must be in the X-register, and **X** must be specified in the Alpha register.

The input specifies matrix name A (or X), matrix name B (or X), result matrix C in Alpha register. The outputs are respectively:

$$\begin{aligned} c_{ij} &= a_{ij} + x \text{ or} \\ c_{ij} &= x + b_{ij} \text{ or} \\ c_{ij} &= a_{ij} + b_{ij} \text{ for all } i, j \text{ in } C. \end{aligned}$$

$$\begin{aligned} c_{ij} &= a_{ij} - x \text{ or} \\ c_{ij} &= x - b_{ij} \text{ or} \\ c_{ij} &= a_{ij} - b_{ij} \text{ for all } i, j \text{ in } C. \end{aligned}$$

$$\begin{aligned} c_{ij} &= a_{ij} \times x \text{ or} \\ c_{ij} &= x + b_{ij} \text{ or} \\ c_{ij} &= a_{ij} \times b_{ij} \text{ for all } i, j \text{ in } C. \end{aligned}$$

$$\begin{aligned} c_{ij} &= a_{ij} \div x \text{ or} \\ c_{ij} &= x \div b_{ij} \text{ or} \\ c_{ij} &= a_{ij} \div b_{ij} \text{ for all } i, j \text{ in } C. \end{aligned}$$

The true matrix multiplication calculates each new element i.j by multiplying the i-th. row in A by the j-th. column in B. The input is the three matrix names in Alpha where C must be different from the two operands A and B. The output is:

$$c_{ij} = \sum_{k=1}^{p} a_{ik} \times b_{kj},$$ where A has p columns and B has p rows.

2.3.2. Major Matrix Operations.

The major matrix operations are: inversion, finding the determinant, transposition, and solving a system of linear equations.

	Function	Description	Input
1	**MDET**	Finds the Determinant of a square matrix	Matrix Name in Alpha
2	**MINV**	Inverts and replaces the square matrix	Matrix Name in Alpha
3	**MSYS**	Solves a system of linera equations	Matrix Name A. Name B in Alpha
4	**MTRPS**	Transposes and replaces the real matrix	Matrix name in Alpha

This is where the Advantage really took the original CCD implementation to its full fulfillment, as the CCD was sorely lacking the major operations - no doubt due to the size constrains in a module that already had tons of other wonders and was packed bursting to its seams.

I recall the awe with which we used to run **MINV** and the other functions: just a single keystroke doing the same as all those intricate FOCAL programs did using Gaussian algorithms, element pivoting and row simplification... simply amazing back then. It was the ultimate Matrix function set, pretty much surpassing the HP-15C implementation in this area. If you're reading this now I suspect you probably had a similar experience too; but enough reminiscing and let's get on with the manual.

The output of these operations always replaces the original matrix with the result. Moreover, for **MDET** and **MSYS the** result matrix is placed in its LU-decomposed form, which makes it not suitable for some direct subsequent operations.

Nole: You cannot transpose or change any element of a matrix A that has had its determinant found or has had its solution matrix found because **MDET** and **MSYS** transform the input matrix A into its LU-decomposed form. (Refer to '"LU-Decomposition" for more information.) However, you can retrieve the original form of A from its decomposed form by inverting it twice (execute MINV twice). The LU-decomposition does not interfere with the ca lculations for **MINV**, **MSYS**, or **MDET**.

Example 1.

Find the determinant of the inverse of the transpose of the matrix :
Storing it in Main Memory, starting in Register R0.

$$\begin{bmatrix} 6 & 3 & -2 \\ 1 & 4 & -3 \\ 2 & 3 & -1 \end{bmatrix}$$

First make sure that the calculator SIZE is set at least to 10 to accommodate the elements plus the header register, typing XEQ "**SIZE**" 010. Next we begin by creating the matrix in main memory, using the name 'R0" in Alpha and the dimension in X:

ALPHA, "**R0**", ALPHA
3.003, XEQ "**MATDIM**"

Since the elements are all integer numbers, this is an ideal candidate for **PMTM**:

XEQ "**PMTM**" , -> at the prompt "R1: _" we type: 6, ENTER^, 3, ENTER^, CHS, 2, R/S
-> at the prompt "R2: _" we type: 1, ENTER^, 4, ENTER^, CHS, 3, R/S
-> at the prompt "R3: _" we type: 2, ENTER^, 3, ENTER^, CHS, 1, R/S

And now the festival begins - type:

XEQ "**TRNPS**",	R0 is transposed
XEQ "**MINV**",	R0 (which was transposed) is inverted
XEQ "**MDET**"	-> 0.040 is the solution.

Note that if you had wantecacad to find the transpose of the original matrix after having found its determinant, you would have needed to invert the matrix twice to change the LU-decomposed form back to the original matrix.

LU-Decomposition

The *lower-upper (LU) decomposition* is an unrecognizably altered form of a matrix, often containing Alpha data. This transformation properly occurs in the process of finding the:

- Solution to a system of equations (**MSYS**; SE in the **MATRX** program).
- Determinant (**MDET**; DT in **MATRX** program).
- Inverse (**MINV**; I in **MATRX** program).

The first two of these operations convert the input matrix to its LU-decomposed form *and leave it there*, whereas inversion leaves the matrix in its inverted form. When you use functions that produce an LU-decomposed form, there are several things that you need to be aware of:

• You cannot **edit** an LU-decomposed matrix unless you edit every element. Also care must be exercised when **viewing** an LU-decomposed matrix. Certain operations can alter elements without your knowledge (refer to "Editing and Viewing an LU-Decomposed Matrix" below for more details).

• You cannot perform any operation that will modify the matrix (other than **MINV**) because the LU status of the matrix will be cleared and it will become unrecognizable. Operations that have this effect are: **R<>R, C<>C**, **MS**, **MSR+**, **MSR-**, **MSC+**, **MSC-**, **MMOVE** (intramatrix), **MSWAP**, and **TRNPS**.

• LU-decomposition destroys the original form of the matrix. So if you perform **MSYS** or **MDET** and then try to look at your input matrix (A in the **MATRX** program), *you will find only the altered, decomposed form.*
• You cannot calculate the transpose (**TRNPS**; [SHIFT][**B**] in **MATRX** program) of a matrix in LU-decomposed form. LU-decomposition *does not hinder the correct calculation of the inverse, determinant, or solution matrix*, since these operations require the LU-decomposition anyway.

Reversing the LU-Decomposition.- To restore a matrix to its original form from its decomposed form, simply *invert it twice* (in effect: find the inverse and then re-invert to the original). Naturally, for this to work the matrix must be invertible (non-singular). The result can differ slightly from the original due to rounding-off during operations.

Editing and Viewing an LU-Decomposed Matrix.- LU-decomposed matrices are stored in a different form than normal matrices:

- Certain elements contain alpha data. (or Non-normalized numbers to be precise)
- The matrix status register is modified to indicate that the matrix is in LU form.

Editing *any* element of the matrix will clear the LU-flag in the status register, which makes the matrix unrecognizable to the program. Because of this, if you edit one element, you must edit them all if you wish to use the matrix again. Note that the matrix will no longer be in LU-decomposed form after this action. You *can* view the contents of an LU-decomposed matrix by doing one of the following:

- From the **MATRX** main menu press [SHIFT][**A**] to view individual elements without modifying them.

- Set flag 08 before executing **MEDIT** or **CMEDIT**. This allows you to view the elements without modifying them.

Header Register X-ray. { LU? }

The graphic below shows the different fields in the Matrix header register (14 bytes in total):

13	12	11	10	9	8	7	6	5	4	3	2	1	0
"4"	File Addr			LU?	*# of Columns*			*Active ij*			File Size		

Note that a matrix file in X-mem has its type set to 4 (in leftmost byte), and that the matrix dimensions can be derived from the information in the file size field (nybbles 0,1,2) and the number of columns field (nybbles 6,7,8), whereby: Number of rows = File size / Number of Columns.

Lastly the pointer field stores the information on the current element as a counter starting from the first element (1) to the last (nxm). Given the length of this field it follows that a maximum of 4,096 elements (FFF) can be tracked, equivalent to a square matrix of dimensions 64 x 64 or any equivalent (m x n) combination.

You can use the function **LU?** to check whether a matrix is in its LU-decomposed form. It'll return YES/NO in Run mode, and in a program will skip the next line when false (i.e. it's NOT decomposed).

Working with Complex Matrices.

When working with complex matrices it is most important to remember that, in the calculator, a complex matrix is simply a real matrix with four times as many elements. Only the **MATRX** program and the complex-matrix editor (**CMEDIT**) "recognjze" a matrix as complex and treat its elements accordingly. All other functions treat the real and imaginary parts of the complex elements as separate real elements.

How Complex Elements are represented

In its internal representation a complex matrix has twice as many columns and twice as many rows as it "normally' would.

The complex number 100 + 200i is stored as $\begin{bmatrix} 100 & -200 \\ 200 & 100 \end{bmatrix}$

The 2 x 1 complex matrix $\begin{bmatrix} 1 + 2i \\ 3 - 4i \end{bmatrix}$ is stored as $\begin{bmatrix} 1 & -2 \\ 2 & 1 \\ 3 & 4 \\ -4 & 3 \end{bmatrix}$

There is one important exception to this scheme: for the column matrix (a vector) in a system of simultaneous equations.

Solving Complex Simultaneous Equations.- The easiest way to work with complex matrices is to use the **MATRX** program. It automatically dimensions, input and output complex matrices. However, **MSYS** can solve more complicated systems of equations than **MATRX** can.

In addition, a complex result-matrix from the **MATRX** program cannot be used for many complex-matrix operations outside of MATRX. This is because **MATRX** will dimension a complex column matrix differently than 2m x 2. Instead, it uses the dimensions 2m X 1, in which the real and imaginary parts of a number become successive elements in a single column.

This form has the advantage of saving memory and speeding up opera tions. The complex-matrix editor and **MSYS** can also use this 2m X 1 form, though they do not require it. This means you can use **MSYS** on a matrix system from **MATRX**. You can convert an existing 2m x 2 complex column matrix to the 2m X 1 form by transposing it, redimensioning it to 1 x 2m, then retransposing it. There is no easy way back.

Accessing Complex Elements.- If you use the complex-matrix editor (**CMEDIT** or the editor in the **MATRX** program), you can access complex elements as if they were actual complex numbers. Otherwise (such as when you use pointer-setting functions), you must access complex elements as real elements stored according to the 2m x 2n scheme given above.

Storage Space in Memory.- Since the dimensions required for a complex matrix are four times greater than the actual number of complex elements (an m X n complex matrix being dimensioned as 2m x 2n), realize that the number of registers a complex matrix occupies in memory is correspondingly four times greater than a real matrix with the same number of elements. In other words, think of a complex matrix's storage size in terms of its **MATDIM** or **DIM?** dimensions, not its number of complex elements.

Using Functions with Complex Matrices

Most matrix functions do not operate meaningfully on complex matrices: since they don't recognize the different parts of a complex number as a single number, the results returned are not what you would expect for complex entries.

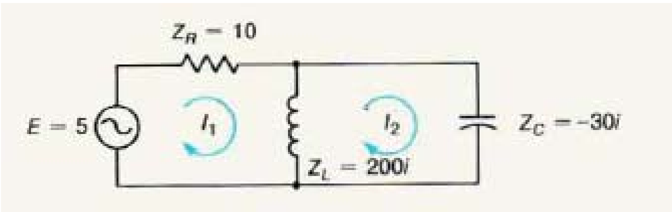
Valid Complex Operations. Certain matrix functions work equally well with real and complex functions. These are:

- **MSYS** Solving simultaneous equatiohs
- **MINV** Matrix inverse
- **MAT+** Matrix add
- **MAT-** Matrix subtract
- **MAT*** Matrix scalar multiply, but only by a real scalar in X-reg.
- **M*M** Matrix multiplication

Both the input and result matrices must be complex.

<u>Example 2.</u>

Engineering student A.C. Dimmer wants to analyze the electrical circuit shown below. The impedances of the components are indicated in complex form. Determine the complex representation of the currents i1 and i2



The system can be represented by the complex matrix equation: AX = B, or

$$\begin{bmatrix} 10 + 200i & -200i \\ -200i & (200 - 30)i \end{bmatrix} \begin{bmatrix} I_1 \\ I_2 \end{bmatrix} = \begin{bmatrix} 5 \\ 0 \end{bmatrix}$$

We'll use the individual matrix functions instead of **MATRX** program, already covered in the previous sections.

The main thing to sort out in this example is the dimension of the matrices involved. The coefficients matrix **A** is a 2 x 2 complex matrix, thus as per the previous paragraphs we will need (4x4 +1) = 17 registers. The independent terms matrix **B** is a 2 x 1 complex matrix, thus will need (4x2 +1) = 9 registers.

This makes for a total of 26 registers needed for the example; therefore we adjust the SIZE accordingly first typing: XEQ 'SIZE" 026.

Next we create the two matrices in main memory, starting at R00 and R17 respectively. Note the shortcut in the R0 name – dropped the zero.

ALPHA, "R", ALPHA	ALPHA, "R17", ALPHA
4.004, XEQ "**MATDIM**"	4.002, XEQ "**MATDIM**"

The next step is entering the element values – using **CMEDIT** because that is the only editor capable of editing complex matrices, as we know.

CMEDIT	RE.1:1= ?	Complex-matrix editor.
10 R/S 200 R/S	RE.1:2= ?	Loads the real and imaginary parts of elements into **R0**, the coefficient matrix (A).
0 R/S 200 CHS R/S	RE.2:1= ?	
0 R/S 200 CHS R/S	RE.2:2= ?	
0 R/S 170 R/S	−170.0000	
ALPHA R17 ALPHA		Dimensions the column matrix **R17** to 4 × 2 for 2 complex rows and 1 complex column. It needs 9 registers.
4.002 MATDIM	4.0020	
CMEDIT	RE.1:1= ?	Complex-matrix editor.
5 R/S 0 R/S	RE.2:1= ?	Loads the real and imaginary parts of elements into **R17**, the column matrix (B).
0 R/S 0 R/S	0.0000	

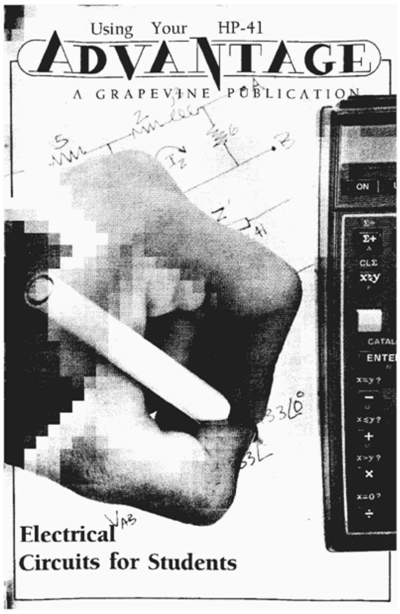
Finally it comes the time for the real work: using **MSYS** to solve the system, and **MCEDIT** again (in view-only mode) to review the results:

Keystrokes	Display	
ALPHA R,R17 ALPHA XEQ MSYS	0.0000	Calculates the solution matrix (X) and loads it into **R17**.
SF 08		Sets editor for view-only operation.
ALPHA R17 ALPHA XEQ CMEDIT	RE.1:1=0.0372	Displays the complex results for I_1 and I_2, which are in **R17**. If you have a printer attached and set flag 08 before executing CMEDIT, all elements will be printed out automatically.
R/S	IM.1:1=0.1311	
R/S	RE.2:1=0.0437	
R/S	IM.2:1=0.1543	

The solution is:

$$\begin{bmatrix} I_1 \\ I_2 \end{bmatrix} = \begin{bmatrix} 0.0372 + 0.1311i \\ 0.0437 + 0.1543i \end{bmatrix}$$

As you can see this is an EE student's dream for circuit analysis – if this is in your area of interests you should check out the macro-program written by Ted Wadman, Chris Coffin and Robert Bloch as one of the proverbial three best examples of utilization of the Advantage Module.



The program is documented in its dedicated Grapevine booklet, available at:

http://www.hp41.org/LibView.cfm?Command=View&ItemID=523

and for further convenience Jean-Francois Garnier put it in ROM module format, available at:

http://www.hp41.org/LibView.cfm?Command=View&ItemID=613

The module also contains the other two famous applications of yore:

1. "Electrical Circuits for Students",
2. "Statics for Students" , and
3. "Computer Sicence on your HP-41" (a.k.a. the HP-16C Emulator).

Anybody curious enough to see what could be done with the Advantage is encouraged to check those out – you'll be rewarded.

The last example asks you to solve a set of six simultaneous equations with six unknown variables. This requires the use of **MSYS**, as the constant matrix *B is not a column matrix*.

Example 3.

Silas Farmer has the following record of sales of cabbage and broccoli forthree different weeks. He knows the total weight of produce sold each week, the total price received each week, and the price per pound of each crop. The price of cabbage is $0.24/kg and the price of broccoli is $0.86/kg. Determine the weights of cabbage and broccoli he sold each week.

	Week-1	**Week-2**	**Week-3**
Combined Weight (kg)	274	233	331
Combined Value	$130.32	$112.96	$151.36

The following set of linear equations describes the two unknowns (the weights of cabbage and broccoli) for all three weeks, where the first row of the constant matrix represents the weights of cabbage for the three weeks and the second row represents the weights of broccoli. Since the constant matrix is not a column matrix, you must use **MSYS** and not the **SE** function in the **MATRX** program.

$$\begin{bmatrix} 1 & 1 \\ 0.24 & 0.86 \end{bmatrix} \begin{bmatrix} d_{11} & d_{12} & d_{13} \\ d_{21} & d_{22} & d_{23} \end{bmatrix} = \begin{bmatrix} 274 & 233 & 331 \\ 120.32 & 112.96 & 151.36 \end{bmatrix}$$

Where the subindices indicate the crop (1= broccoli, 2=cabbage), and the week (1,2,3), and the first row describes the weight equations, and the second the prices relationship.

Calling "FACTORS" the coefficients matrix and "LINKS" the costant matrix, we first create them by dimensioning in X-Memory as follows:

ALPHA, "FACTORS", ALPHA,
2.002, XEQ "**MATDIM**"

ALPHA, "LINKS", ALPHA,
2.003, XEQ "**MATDIM**"

Next we'll use **PMTM** to input all the element values. Note that even the "longest" row has 20 characters (including the separator blanks), which is below the limits of the ALPHA register length, of 24 characters max.

With "FACTORS" in Alpha we type:

XEQ"**PMTM**" -> at the prompt "R1: _" we type: 1, ENTER^, 1, R/S
-> at the prompt "R2: _" we type: 0, [,], 2, 4, ENTER^, 0, [,], 8, 6, R/S

With "LINKS" in Alpha we type:

XEQ "**PMTM**" -> at the prompt "R1: _" we type: 2,7,4, ENTER^, 2,3,3, ENTER^, 3,3,3, R/S
-> at the prompt "R2: _ " we type: 1,2,0,[,],3,2, ENTER^, 1,1,2,[,],9,6, ENTER^, 1,5,1,[,],3,6, R/S

All set up we simply execute **MSYS** to obtain the solutions shought for:

ALPHA, "FACTORS,LINKS", ALPHA
XEQ "**MSYS**"

	Week-1	**Week-2**	**Week-3**
Cabbage Weight (kg)	186	141	215
Broccoli Weight (kg)	88	92	116

Note: using **OMR** (or **OMC**) to output the elements of the matrix B you can see how the results are all *integer values* – which speaks of the accuracy of the internal operaions, taking advantage of the 13-digit math routines available in the OS for MCODE.

Note also that with these programs the integer results are shown without any zeros after the decimal point, regardless of the current display settings (FIX or otherwise).

OMR and **OMC** are extension functions – pretty much like **PMTM** is - and will be described in detail in chapter 3.

2.3.3.- Other Matrix Functions ("Utilities")

The remaining matrix functions, also called utilities, are those for copying and exchanging parts of matrices, and miscellaneous, extra arithmetic functions: finding sums, norms, maxima, and minima, and matrix reduction.

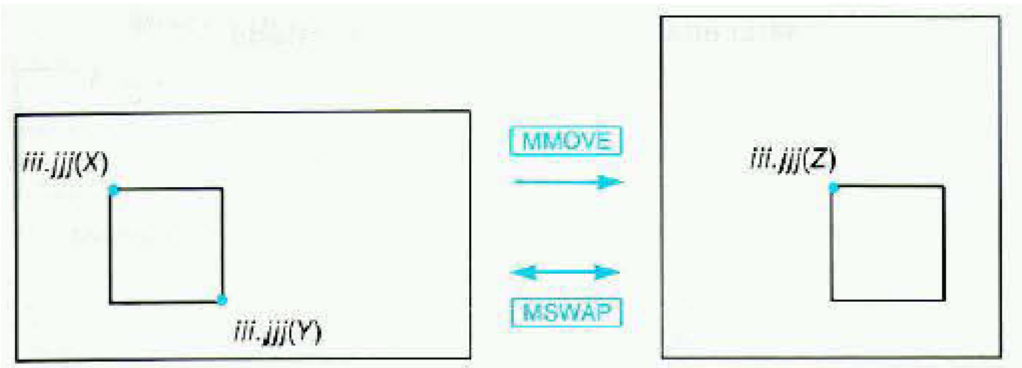
Moving and Exchanging Matrix Sections.

	Function	Description	Input
1	**C<>C**	Exchange columns k and l in a matrix	Name in Alpha, kkk.lll in X-reg
2	**R<>R**	Exchange Rows k and l in a matrix	Name in Alpha, kkk.lll in X-reg
3	**MMOVE**	Matrix Move	Names in Alpha, Pointers in stack
4	**MSWAP**	Matrix Swap	Names in Alpha, Pointers in stack

MMOVE and **MSWAP** Copies or Exchanges the submatrix defined by pointers in the source matrix to the area defined by one pointer in the target matrix. The inputs require both matrix names in Alpha separated by a comma, plus the pointers in the stack as follows:

in X-reg: *iii.jjj* for *A*'s initial element;
in Y-reg: *iii.jjj* for *A*'s final element;
in Z-reg: *iii.jjj* for *B*'s initial element.

When executing **MMOVE** and **MSWAP** if **A** and **B** are the same matrix and the source submatrix overlaps the target submatrix, the elements are processed in the following order: reverse column order (last to first) and reverse element order (last to first) within each column.



Source matrix (*A*)   Target matrix (*B*)

When an input of the form iii.jjj is expected in the X-register, a zero value for either the i-part or the j-part is interpreted as 1. (Zero alone equals 1.001.) This is true for the iii.ijj-values that MMOVE and MSWAP expect in the X- and Z-registers, but *not for the pointer value in the Y-register*.

For the Y-register input, a zero value for the i-part is interpreted as m, the last row, while a zero value for the j-part is interpreted as n, the last column. This convention facilitates easy copying (or exchanging) of entire matrices because simply by clearing the stack (**CLST**) or entering three zeros you specify the elements 1.001 (X) and mmm.nnn (Y) for the first matrix and element 1.001 (Z) for the second matrix, thus defining two entire matrices.

For example in a 4 x 5 matrix:

Y-Register	Pointer Value
0.000	4.005
3.000	3.005
0.003	4.003

Miscellaneous Arithmetic Functions: Maxima and Minima

	Function	Description	Input / Output
5	**MAX**	Finds the maximum element in matrix. Sets element pointer to it.	Matrix Name in Alpha. Outputs element value to X-reg
6	**MIN**	Finds the minimum element in matrix. Sets element pointer to it.	Matrix Name in Alpha Outputs element value to X-reg
7	**MAXAB**	Like MAX but in absolute value. Sets element point to it.	Matrix Name in Alpha Outputs element value to X-reg
8	**CMAXAB**	Finds maximum absolute value in k-th. column. Sets element pointer to it.	Matrix name in Alpha, kkk in X-reg. Outputs element value to X-reg
9	**RMAXAB**	Finds maximum absolute value in k-th. row. Sets element pointer to it.	Matrix name in Alpha, kkk in X-reg. Outputs element value to X-reg

Miscellaneous Arithmetic functions: Norms and Sums

	Function	Description	Input / Output
10	**CNRM**	Column Norm. Finds the largest sum of the absolute values of the elements in each colum of matrix.	Matrix name in Alpha. Outputs colum norm to X-reg. Sets pointer to first element of colum.
11	**FNRM**	Frobenius Norm. Calculates the square root of the sum of the squares of all elements in matrix.	Matrix name in Alpha. Outputs frobenius norm into X-reg
12	**RNRM**	Row Norm. Finds the largest sum of the absolute values of the elements in each row of matrix.	Matrix name in Alpha. Outputs row norm to X-reg. Sets pointer to first element of row.
13	**SUM**	Sums all elements in matrix.	Matrix name in Alpha. Outputs the sum to X-reg
14	**SUMAB**	Sums absolute values of all elements in matrix.	Matrix name in Alpha Outputs the sum to X-reg
15	**CSUM**	Finds the sum of each column and stores them in a result vector.	Matrix name , result matrix name (Vector) in Alpha. (*)
16	**RSUM**	Finds the sum of each row and stores the sums in a result vector.	Matrix name , result matrix name (Vector) in Alpha. (*)

(*) For **CSUM** and **RSUM** the number of elements in the result matrix (vector) must equal the number of columns/rows in the input matrix.

Miscellaneous Arithmetic functions: Matrix Reductions

	Function	Description	Input / Output
17	**YC+C**	Multiplies each element in column k of matrix by value in Y-ref. and adds it to corresponding element in column l	Matrix name in Alpha, kkk.lll in X-reg, y in Y-reg. It changes the elements in colum l
18	**PIV**	Finds the pivot value in column k, that is the maximum absolute value of an element on or below the diagonal.	Matrix Name in Alpha, kkk in X-reg
19	**R>R?**	Compares elements in rows k and l. If (and only if) the first non-equal element in k is greater than its corresponding element in l, then the comparison is positive for the "do if true" rule of programming.	Matrix name in Alpha, kkk.lll in X-reg Outputs "YES" if first non-equal element in row k is greater than element in row l. "NO" in all other case.

The last two functions are not operating on a matrix, but are auxiliary for the FOCAL programs:

	Function	Description	Input / Output
20	**AIP**	Appends the absolute value of the integer part of the number in X to the contents of the Alpha register.	Value in X.
21	**MPT**	Appends a matrix prompt "rrr.ccc=" to the contents of the Alpha register (dropping leading zeros in each part)	rrr.ccc in X-reg

Note that **AIP** and **AINT** in the SandMath are very similar – but **AINT** won't take the absolute value. This fact is useful to append integer vaules to alpha without decimal numbers, but respecting the sign.

Note that **MPT** in the SandMatrix is an enhanced version written in MCODE – that replaces the mini-FOCAL program used in the Advantage.

Example. Calculate the Row, Column and Frobenius norms for the matriw

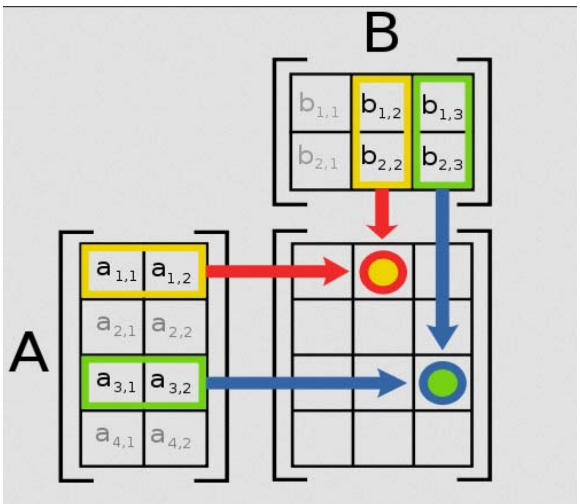
$$A = \begin{bmatrix} 3 & 5 & 7 \\ 2 & 6 & 4 \\ 0 & 2 & 8 \end{bmatrix}, \qquad \|A\|_F = \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n} |a_{ij}|^2}$$

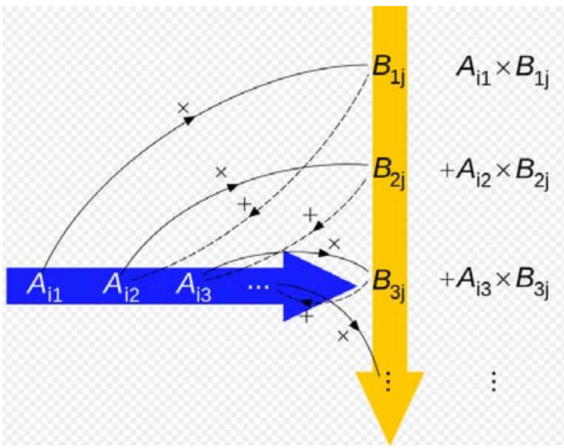
$$\|A\|_1 = \max_{1\le j\le n} \sum_{i=1}^{m} |a_{ij}|,$$ which is simply the maximum absolute column sum of the matrix.

$$\|A\|_\infty = \max_{1\le i\le m} \sum_{j=1}^{n} |a_{ij}|,$$ which is simply the maximum absolute row sum of the matrix

The results are:	Row Norm	= 19
	Column Norm	= 15
	Frobenius Norm	= 14,38749457

The Frobenius norm will come very handy for some programs in Chapter-3 as convergence criteria, and to determine whether two matrices are "equivalent" in reduction algorithms.





3. Upper-Page Functions in detail

This chapter is all above and beyond the matrix functionality present in the Advantage Pac – a true extension of its capabilities into new and often uncharted territories.

3.1. ADV MATRIX

3.1. The Enhanced Matrix Editor(s)

Often the most tedious part of a matrix calculation becomes the data entry for the input matrices and the review of the results. With this in mind the SandMatrix includes convenient alternatives to **MEDIT**, the "standard" Matrix Editor from the Advantage, seen in the previous chapter. There are as follows:

	Function	Description	Input / Output
1	**PMTM**	Prompt Matrix by Rows	Matrix name in Alpha
2	**IMR**	Input Matrix by Rows	Matrix name in Alpha
3	**IMC**	Input Matrix by Columns	Matrix name in Alpha
4	**OMR**	Output Matrix by Rows	Matrix name in Alpha
5	**OMC**	Output Matrix by Column	Matrix name in Alpha
6	**OXC**	Output Column k	Matrix name in Alpha, kkk in X-reg
7	**OXR**	Output Row k	Matrix name in Alpha, kkk in X-reg

Of all these more remarkable one is of course **PMTM** – which expedites element data entry to the maximum possible in the 41 platform, almost as if it were a full-fledge editor in a graphical screen.

The idea is to use the Alpha register as repository for all the elements, separating the individual values by spaces (entered using the **ENTER^** key). The data input is terminated by presing R/S.

The back arrow key is always active to correct a wrong entry, and will terminate the function if Alpha is completely cleared. **PMTM** allows for negative and decimal numbers to be entered, thus the **CHS** and **RADIX** keys are also active during the data entry prompt. Furthermore, the logic will only allow one occurrence of these per each element within the prompt string.

PMTM knows how many rows should be input (it is part of the matrx dimension), thus the prompts will continue to appear until the last row is completed. A row counter is added to the promt to indicate the current row being edited.

If you enter fewer elements in the prompt than existing columns, the remaing elements will be left unchanged and the execution will end. Conversely, if you enter more elements in the prompt than existing columns, those exceeding the quota (the extra ones) will simply be ignored.

The two limitations of PMTM are as follows:

- A maximum length of 24 characters is possible during the prompt. This includes the blank separators, the comma (radix), and the negative signs if present.
- No support for the Exponential format is implemented (EEX). You need to use any of the other editors if your element values require such types of data.

Obviously this makes **PMTM** ithe ideal choice for matrices containing integer numbers as elements – but not exclusively so as it can also be used for other values (real-numbers) as long as the two condicions above are respected.

At the heart of **PMTM** there is is function **^MROW** ("Enter Matrix Row"), responsible for the presentation of the prompt in Alpha and accepting the keyboard pressings there to make up the string (or list) with all values. It also provides the logic of actions for the control keys, like ENTER^, Back arrow, R/S, etc.

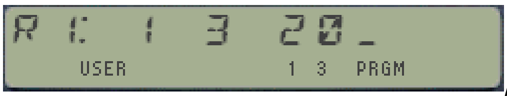
^MROW is called in a loop as many times as rows exist in the matrix, while **ANUMDL** (in the SandMath) is used every iteration (each time a row is being processed) to "extract" the individual element data from the global string in the prompt.

Below is the program listing for **PMTM**, and as you can see it's just a sweet & short driver for **^MROW** that also takes advantage of the auxiliary functions in the SandMatrix.

1	LBL "PMTM"	
2	0	
3	MSIJA	*position pointer to 1.1*
4	LBL 01	
5	MRIJ	*recall pointer*
6	INT	*row number*
7	^MROW	*prompts for string*
8	CF 22	*default reset*
9	LBL 00	*separate elements*
10	ANUMDL	
11	FC?C 22	*last one reached?*
12	GTO 02	*yes, exit*
13	MSR+	*store element*
14	FC? 09	*end of row?*
15	GTO 00	*no, do next element*
16	FC? 10	*end of matrix?*
17	GTO 01	*no, do next row*
18	LBL 02	
19	MNAME?	*recall Mname*
20	END	*done.*

^MROW is the first function listed in CAT"2 within the "-ADV MATRIX" group – and rightfully so. Note that even if **PMTM** is not strictly an MCODE function, de-facto it is a hybrid one, and therefore it's denoted in blue color all throughout this manual. If **PMTM** is the beauty then **^MROW** is the beast. If you're interested you can review the MCODE listings for it in appendix "M".

Below are two examples of the lists being edited, for the first two rows of a given matrix:



, and

The built-in logic allows for *just one negative sign and one radix character per each value entry*.

Note that **^MROW** is also used by **PMTP**, the "Polynomial Input" function, which has a very parallel structure to **PMTM** and is used to enter the coefficients of a polynomial into data registers. It will be covered in the polynomial section later on.

The remaining routines in this section all deal with Input and Output of the matrix elements, depending on whether it's done following the Row or Column sequence, as well as two functions to only view one specific row or column (**OXR** and **OXC**).

They are very much equivalent to **MEDIT** in many aspects, although the symbol "**a**" is used in the prompts. They are slightly faster and offer the added convenient feature that for *integer element values* the zeros after the decimal point are not shown in the prompt – regardless of the current display settings (FIX or otherwise). This makes for a clearer UI.

The program listing is shown below; note how the different entry points set the appropriate subset of user flags, and that they all share the main section for tha actual element input and review.

1	LBL "OMR"		33	MSIJA	*set pointer to row/col*
2	0	*clears F0-F7*	34	LBL 00	
3	*GTO 05*		35	*"a"*	*element symbol*
4	LBL "OMC"		36	MRIJ	*recall index*
5	2	*sets F1*	37	MP	*prompt index*
6	*GTO 05*		38	MR	*recal value*
7	LBL "IMR"		39	FS? 04	*LU decomposed?*
8	E	*sets F0*	40	*GTO XX*	*synthetic jump (!)*
9	*GTO 05*		41	INT?	*integer?*
10	LBL "IMC"		42	AINT	*yes, append IP*
11	3	*sets F0 & F1*	43	FRC?	*fractional?*
12	LBL 05		44	ARCL X	*yes, append all*
13	X<>F		45	FC? 00	*view only?*
14	LU?	*is LU decomposed?*	46	AVIEW	*yes, show it*
15	SF 04	*yes, flag this fact*	47	FC? 00	*view only?*
16	0		48	*GTO 02*	*yes, skip editing*
17	MSIJA	*resets pointer to 1:1*	49	*"|-?"*	*append "?"*
18	*GTO 00*	*go to first element*	50	PROMPT	*show current value*
19	LBL "OXC"		51	MS	*store new value*
20	E1	*sets F1 & F3*	52	LBL 02	
21	GTO 04		53	FC? 01	*by column?*
22	LBL "OXR"		54	J+	*yes, next column*
23	8	*sets F3*	55	FS? 01	*by row?*
24	LBL 04		56	I+	*yes, increase row*
25	X<>F		57	E1	*F10*
26	LU?	*is LU decomposed?*	58	FS? 03	*by row?*
27	SF 04	*yes, flag this fact*	59	DSE X	*yes, F9*
28	RDN		60	FC? IND X	*end of matrix/row?*
29	INT		61	*GTO 00*	*no, next element*
30	E3/E+		62	MNAME?	*yes, recall Mname*
31	FC? 01	*row?*	63	END	*and end.*
32	I<>J	*yes, transpose*			

Other pointer utilities included are listed in the table below; they are used in many of the FOCAL programs described in the following sections.

	Function	Description	Input / Output
8	**^MROW**	Prompts the list and controls input	Element values as Alpha List
9	**I<>J**	Swaps iii and jjj in X (also does E3/ for integers)	iii.jjj in X-reg. Index swapped to jjj.iii
10	**I#J?**	Tests whether iii is different from jjj	iii.jjj in X. YES/NO, do if true.
11	**SQR?**	Tests for Square Matrices	MNAME in Alpha. YES/NO, do if True..
12	**MFIND**	Finds an element in a given matrix and sets element pointer to it	Element value in X-reg Outputs the pointer iii/jjj to X-reg

3.2. New Matrix Math functions.

3.3.1. Utility / housekeeping functions: rounding the capabilities.

This group comes very handy for the handling and management of intermediate steps required as part of more complex algorithms. As a rule, the functions work for matrices stored either in main memory or in X-memory. Only **MATP** and **MAT=** create new matrices; all other functions expect them already dimensioned.

	Function	Description	Input / Output
1	**MAT=**	Makes matrix B equal to A B = A	Matrix names in Alpha: "A,B". Both must exist.
2	**MATP**	Driver for M*M operation	Under program control. Creates both matrices on the fly.
3	**MCON**	Matrix from a constant Makes aij = x, i=1,2,..m; j=1,2,..n	Matrix name in Alpha, constant in X-reg Makes all matrix elements equal to x
4	**MFIND**	Finds an element within a matrix	Matrix Name in Alpa, element in X-reg. Returns pointer to X and set to element.
5	**MIDN**	Makes identity Matrix Makes aii =1 and aij=0 for i#j	Matrix name in Alpha. (must exist)
6	**MRDIM**	Re-dimensions Matrix (properly) It keeps existing elements in place.	Matrix name in Alpha, dimension in X. Output is a new matrix (adds ' to name)
7	**MSORT**	Sorts all elements within a matrix	Matrix Name in Alpha. Reorders elements in ascending order.
8	**MSZE?**	Calculates the Matrix size Size = m x n	Matrix name in Alpha. Output is placed into X-reg.
9	**MZERO**	Zeroes (clears) all elements in matrix Makes aij = 0, i=1,2..m; j=1,2,..n	Matrix name in Alpha All elements are set to zero.

A few remarks on each of these functions follow, as well as the program listings.

MAT= copies an existing matrix into another, with names in Alpha. Prior to doing the bulk element copy, it redimensions the target matrix to be the same as the source one. *It is however not required that the target matrix already already exist* – it will be created if not already there.

MCON does a simple thing: converts the value in the X-Reg into a matrix with all elements equal to this value. This is useful in some calculations and for matrix manipulations. See the simple program listings for these routines below;

1	LBL "MAT="	***"A,B" expected in Alpha***	1	LBL "MCON"	*MNAME in Alpha*
2	DIM?	*dimension*	2	MZERO	*clear all elements*
3	ASWAP	*swap Alpha*	3	RDN	*get constant back to X*
4	MATDIM	*re-dimension target*	4	"X"	*prepare alpha string*
5	ASWAP	*undo the swap*	5	MAT+	*add x to all elements*
6	CLST	*prepare pointers*	6	MNAME?	*recall MNAME to Alpha*
7	MMOVE	*move all elements*	7	END	*done*
8	END	*done*			

MZERO is the unsung hero behind other routines – as the proper way to clear a matrix file, since **CLFL** cannot be used because it also clears the header register (it was meant for Data files). Use it safely for matrices in main and x-memory.

MSORT uses an auxiliary matrix in main memory ("**R0**") where **RGSORT** (from the SandMath) is applied to; then data are copied back to the original matrix. It also checks for available registers, adjusting the calculator SIZE if necessary. The contents of those (n x m +1) data registers will be lost.

1	LBL "MSORT"	*MName in Alpha*	1	LBL "MZERO"	*MNAME in Alpha*
2	SIZE?	*current SIZE*	2	DIM?	*get dimension*
3	MSZE?	*matrix size*	3	SF 25	
4	E		4	PURFL	*purge file*
5	+	*plus one*	5	FC?C 25	*was in main mem*
6	X>Y?	*is it larger?*	6	GTO 01	*jump over*
7	PSIZE	*yes, adjust size*	7	MATDIM	*re-create file*
8	"|-,R"	*prepare Alpha string*	8	RTN	*done*
9	MAT=	*make matrix R0 equal*	9	LBL 01	
10	MSZE?	*its size again*	10	ANUM	*get first reg from title*
11	E3/E+	*prepare control word*	11	ENTER^	*copy in Y-reg*
12	RGSORT	*sort registers*	12	MSZE?	*get matrix size*
13	ASWAP	*swap alpha*	13	+	*add to first reg*
14	CLST	*prepare pointers*	14	E3/3+	*prepare index format*
15	MMOVE	*move all elements*	15	+	*add to first reg*
16	MNAME?	*recall original name*	16	CLRGX	*clear registers*
17	END	*done*	17	END	*done*

MSZE? has a new MCODE implementation in this revision – directly reading the matrix header register. Its funcionallity is equivalent to **FLSIZE** for matrices stored in X-mem – but not so for matrices stored in main memory.

1	MSZE?	*Header*	A60A	*0BF*	**"?"**	
2	MSZE?	*Header*	A60B	*005*	**"E"**	***Matrix Size?***
3	MSZE?	*Header*	A60C	*01A*	**"Z"**	
4	MSZE?	*Header*	A60D	*013*	**"S"**	
5	MSZE?	*Header*	A60E	*00D*	**"M"**	*Ángel Martin*
6	MSZE?	**MSZE?**	**A60F**	*379*	*PORT DEP:*	***Jumps to Bank_2***
7	MSZE?		A610	*03C*	*XQ*	*adds "4" to [XS]*
8			A611	*1D9*	*->A5D9*	*[LNCH0]*
9	*valid for main and X-mem*		A612	***388***	***<parameter>***	*B788*
10	*the proper way to do it!*		A613	*00B*	*JNC +01*	
11			A614	**100**	**ENROM1**	*restore bank-1*
12	MSZE?		A615	0B0	C=N ALL	*header register*
13	MSZE?		A616	106	A=C S&X	
14	MSZE?		A617	*17D*	*?NC GO*	*[BIN-BCD] plus [RCL]*
15	MSZE?		A618	*0C6*	*->315F*	*[ATOX20]*

PMAT is nothing more than a user-friendly driver program to automate the complete matrix product procedure, without any need to dimension the result matrix in advance. The routine will guide you step-by-step thru the complete sequence, including the element data input and output.

MIDN is a good example of a sorely missing function: the majority of matrix algorithms involve identity matrices, one way or another, so having a routine that does the job becomes rather important. The SandMatrix routine follows a *single-element approach*, storing ones in the main diagonal after zeroing the matrix first. This is faster and more convenient that block-based methods, even if not requiring scratch matrices for intermediate calculations. See an the example below courtesy of Thomas Klemm:

```
                     DIM(n+1, n)     A^T          DIM(n, n)

                        |1 1 1|       |1 0 0 0|        |1 0 0|
|1 1 1|      ->         |0 0 0|  ->   |1 0 0 0|   ->   |0 1 0|
                        |0 0 0|       |1 0 0 0|        |0 0 1|
                        |0 0 0|
```

1	LBL "MIDN"	*MNAME in Alpha*	1	LBL "IDN2"	
2	MZERO	*clear all elements*	2	DIM?	*current dimension*
3	0		3	ENTER^	*push it to Y*
4	MSIJA	*set pointer to 1:1*	4	INT	*n*
5	E	*element value*	5	MATDIM	*row matrix, n x 1*
6	LBL 00		6	E	
7	MSC+	*store and increase I*	7	MCON	*all ones*
8	FC? 09	*end of row?*	8	X<>Y	*n*
9	J+	*yes, next row*	9	+	*n+1*
10	FC? 10	*end of matrix?*	10	LASTX	*n*
11	GTO 00	*no, next element*	11	I<>J	*0,00n*
12	END	*yes, done*	12	+	*(n+1),00n*
			13	MATDIM	
shorter and faster, even if more pedestrian			14	TRNPS	
			15	X<>Y	*original n x n*
			16	MATDIM	*bach to shape*
			17	END	*done*

Of all these perhaps only **MRDIM** needs further explanation. Contrary to **MATDIM**, a proper re-dimensioning should respect the elements in the re-dimensioned matrix that held the same position in the original one. **MRDIM** does this, deleting the discarded elements when the redimensioned sub-matrix is smaller than the original, and completing the new onew with zeroes wihen it is bigger (super-matrix). It always starts with a11 (no random origin is possible).

1	LBL "MRDIM"	***MNAME in Alpha***	16	X<>Y	*min(j1,j2)*
2	DIM?	*get dimension*	17	RCL Z	
3	X<>Y	*new dimension to X*	18	INT	*min (I)*
4	ASTO T	*temporary safekeep*	19	+	*min (I), min(j)*
5	*"|-' "*	*add tilde*	20	0	
6	MATDIM	*create new matrix*	21	STO Z	*prepare pointers*
7	CLA		22	ASTO T	*temporary safekeep*
8	ARCL T	*MNAME*	23	*"|-,"*	
9	X>Y?		24	ARCL T	*MNAME*
10	X<>Y	*min(i1,i2)*	25	*"|-' "*	*prepare Alpha string*
11	STO Z	*keep in Z*	26	MMOVE	*copy elements*
12	FRC		27	PURFL	*purge original file*
13	X<>Y		28	MNAME?	*recall name to Alpha*
14	FRC		29	END	*done*
15	X>Y?				

A logical enhancement to this routine would be to change the matrix name back to its original one, removing the tilde. This can be done in two ways:

1. creating a new matrix file and copying it over once again, or (preferable)
2. using **RENMFL** (in the AMC_OS/X module) to rename the X-mem file

Finding an element within a Matrix { MFIND } - plus an easy-driver for M*M

MFIND will search a given matrix looking for an element that equals the value in the X-register. If it is found it returns its location pointer to the X-reg (and leaves the pointer set to it). If it's not found, it returns -1 to X and the pointer is outside the matrix.

You can further use this result adding the conditional test function "**X>=0?**" (available in the SandMath) right after **MFIND** - which in a program will skip a line if the element wasn't found.

Below are the program listings for your perusal.

1	LBL "MFIND"	*MNAME in Alpha*
2	0	
3	MSIJA	*sets pointer to 1:1*
4	LBL 05	
5	RDN	*target value to X-reg*
6	MR	*recall element*
7	X=Y?	*equal?*
8	GTO 02	*yes, exit*
9	J+	*no, increase column*
10	FC? 10	*end of matrix?*
11	GTO 05	*no, next element*
12	RDN	*target value to X-reg*
13	CLX	
14	-	
15	E	*put -1 in X*
16	GTO 00	*exit*
17	LBL 02	
18	RDN	
19	CLX	
20	MRIJA	
21	LBL 00	
22	END	*done*

1	LBL "MATP"	
2	*"DIM1=?"*	*M1 dimension*
3	PROMPT	*prompt for it*
4	*"M1"*	*matrix name - M1*
5	MATDIM	*create matrix in X-mem*
6	PMTM	*input elements*
7	*"DIM2=?"*	*M2 dimension*
8	PROMPT	*prompt for it*
9	*"M2"*	*matrix name - M2*
10	MATDIM	*create matrix in X-mem*
11	PMTM	*input elements*
12	DIM?	
13	FRC	*# of columns for M2*
14	*"M1"*	
15	DIM?	
16	INT	*# of rows for M1*
17	+	*cresult matrix dimension*
18	*"M*"*	*matrix name - M**
19	MATDIM	*create matrix in X-mem*
20	*"|-M1,M2,"*	*prepare Alpha string*
21	2	
22	AROT	
23	M*M	*matrix product*
24	ASHF	*remove acratch*
25	OMR	*output values*
26	END	*done*

Note that in **MATP** I have chosen **PMTM** to enter the element data values – therefore it's somehow limited by the same constraints described before, ie. total length in Alpha and no support for the EEX key.

3.2.2. New Math functions.- Completing the core function set.

The next group incudes advanced application areas in "core" matrix math.

	Function	Description	Input / Output
9	**M^1/X**	Brute-force Matrix X-th Root A = exp(1/x * Ln[A])	Matrix name in Alpha, order in X The result matrix replaces the input
10	**M^2**	Square power of a square Matrix A = [A]^2 = [A].[A]	Matrix name in Alpha The result matrix replaces the input
11	**MDPS**	Matrix Diagonal Product Sum MDPS = Σ[aii*aii+1], i=1,2…n	Matrix name in Alpha. Output is result in X-reg
12	**MEXP**	Exponential of a Matrix A = exp(A)	Matrix name in Alpha. The result matrix replaces the input.
13	**MLIE**	Matrix Lie Product C = AB – BA	Matrix names in Alpha: "A,B,C" Result matrix C must be different.
14	**MLN**	Matrix Logarithm A= Ln (A)	Matrix name in Alpha. The result matrix replaces the input.
15	**MPWR**	Matrix Power of integer order A = A^x	Matrix name in Alpha, order in X-reg. The result matrix replaces the input.
16	**MSQRT**	Matrix Square Root A = sqrt(A)	Matrix name in Alpha. The result matrix replaces the input.
17	**MTRACE**	Calculates the Trace of a Square Matrix Trace = Σ aii, i= 1, 2,..m	Matrix name in Alpha. Output is put into W-reg.
18	**R/aRR**	Row division by diagonal element akj = akj / akk , j= 1,1,…n	Matrix name in Alpha, row kkk in X-reg All row elements divided by akk
19	**ΣIJJI**	Sum of crossed-elements products SCEP = Σ[Σ(aij * aji)]	Matrix name in Alpha Output is put in X-reg.

Formulae and algorithms used.

The algorithms used impose some restrictions to the matrices. These are generally not checked by the programs, thus in some instances there won't converge to a solution. Suffice it to say that the programs are not fool-proof, and assume the user has a general understanding of the subjects – so they won't be used foolishly.

Matrix Exponential { $\boxed{\text{MEXP}}$ }

In mathematics, the matrix exponential is a matrix function on square matrices analogous to the ordinary exponential function. Let **X** be an n×n real or complex matrix. The exponential of **X**, denoted by e^**X** or exp(**X**), is the n×n matrix given by the power series

$$e^X = \sum_{k=0}^{\infty} \frac{1}{k!} X^k$$

where **X**^0 is the identity matrix, **I**. The above series always converges, so the exponential of **X** is well-defined. Note that if **X** is a 1×1 matrix the matrix exponential of **X** is a 1×1 matrix consisting of the ordinary exponential of the single element of **X.**

Finding reliable and accurate methods to compute the matrix exponential is difficult, and this is still a topic of considerable current research in mathematics and numerical analysis. The SandMath uses a direct approach, so no claims of discovering new algorithms"

$$\exp(A) = I + A + A^2/2! + A^3/3! + \ldots\ldots + A^k/k! + \ldots.$$

The program adds new terms until their contribution is negligible, i.e. it results in the same matrix after addng it. This by itself poses an interesting question: how to check whether two matrices are the same? Obviously doing it element-to-element would be a long and impractical method. The alternative is to use the matrix Frobenius norm as a surrogate criterion; assuming that *for very similar matrices*, they'll be equal when they have the same norm.

There's no saying to the execution time or whether the calculator numeric range will be exceeded in the attempt – so you can expect several iterations until it converges. The matrix norm will be displayed after each iteration, so you'll have an indication of the progress made comparing two consecutive values.

Logarithm of a Matrix { MLN }

In mathematics, a logarithm of a matrix is another matrix such that the matrix exponential of the latter matrix equals the original matrix. It is thus a generalization of the scalar logarithm and in some sense an inverse function of the matrix exponential. Not all matrices have a logarithm and those matrices that do have a logarithm may have more than one logarithm. Furthermore, many real matrices only have complex logarithms – making it so even more challenging.

The SandMatrix uses the following algorithm:

If || A - I || < 1, the logarithm of a n x n matrix **A** is defined by the series expansion:

$$\mathrm{Ln}(A) = (A - I) - (A - I)^2/2 + (A - I)^3/3 - (A - I)^4/4 + \ldots\ldots$$ where I is the identity matrix.

Example 1- Calculate the exponential of the matrix **A** given below, and then calculate its logarithm to see how the result matrix compares to the original.

$$\mathbf{A} = \begin{bmatrix} 1 & 2 & 3 \\ 0 & 1 & 2 \\ 1 & 3 & 2 \end{bmatrix}$$

The first part of the assignment is rather simple: Executing MEXP results in the following matrix:

$$\exp(\mathbf{A}) = \begin{bmatrix} 19.45828375 & 63.15030507 & 66.98787675 \\ 8.534640269 & 32.26024414 & 33.27906416 \\ 16.63953207 & 58.45323648 & 61.70173665 \end{bmatrix}$$

However trying to calculate the logarithm will not work, because exp(**A**) doesn't satisfy the requirement: Det[exp(**A**)-I] = -52,95249156; therefore trying **MLN** on it will eventually reach an "OUT OF RANGE" condition.

Example 2.- Calculate the Logarithm of the following matrix:

$$\mathbf{A} = \begin{bmatrix} 1.2 & 0.1 & 0.3 \\ 0.1 & 0.8 & 0.1 \\ 0.1 & 0.2 & 0.9 \end{bmatrix}$$

In this example, || A - I || = 0.5099... < 1 , thus the program will work.

The result matrix after executing **MLN** is as follows:

$$\mathrm{Ln}(\mathbf{A}) = \begin{bmatrix} 0.167083396 & 0.069577923 & 0.287707999 \\ 0.097783005 & -0.240971674 & 0.103424021 \\ 0.086500972 & 0.235053124 & -0.131906636 \end{bmatrix}$$

So we see that unfortunately the logarithm is not a trivial exercise. The programs are listed below, note the combination of both exponential and logarithm into a single program, with flag 01 controlling the case.

1	LBL "MLN"	
2	SF 01	exp flag
3	GTO 00	
4	LBL "MEXP"	
5	CF 01	LN flag
6	LBL 00	
7	DIM?	get dimension
8	I#J?	not square?
9	-ADV MATRX	error message
10	ASTO 01	
11	"|-,^"	
12	MAT=	safekeeping copy
13	DIM?	get dimension
14	"P"	
15	MATDIM	auxiliary matrix
16	"#,"	
17	MATDIM	auxiliary matrix
18	MIDN	
19	ARCL 01	
20	FS? 01	LN?
21	ASWAP	yes, swap names
22	"|-,^"	
23	FS? 01	LN?
24	MAT-	
25	FC? 01	exp?
26	MAT+	
27	"^,"	
28	FNRM	initial norm
29	STO 00	store in R00
30	FC? 01	exp?
31	CLA	
32	ARCL 01	
33	FC? 01	exp?
34	GTO 04	
35	MAT=	
36	CLAC	
37	ABSP	
38	LBL 04	
39	"|-,#"	
40	CLST	
41	MMOVE	
42	2	
43	STO 02	
44	LBL 02	
45	VIEW 00	
46	"#,"	
47	ARCL 01	
48	"|-,P"	
49	M*M	
50	"P,#"	
51	CLST	
52	MMOVE	
53	RCL 02	
54	FC? 01	exp?
55	FACT	to be used as divisor
56	FC? 01	exp?
57	GTO 04	
58	ENTER^	
59	ENTER^	
60	E	to be used as divisor
61	+	
62	CHSYX	
63	LBL 04	
64	"P,X"	
65	MAT/	divide by scalar
66	ABSP	remove "X"
67	" |-^,^"	prepare new string
68	MAT=	safekeeping copy
69	E	
70	ST+ 02	increase term index
71	"^,"	
72	FNRM	new frobenius norm
73	X<> 00	swao with old norm
74	RCL 00	recall new again
75	X#Y?	are the different?
76	GTO 02	yes, keep at it
77	ARCL 01	no, we're done
78	MAT=	
79	PURFL	purges "^"
80	"P,#"	
81	PURFL	purges "P"
82	ASWAP	
83	PURFL	purges "#"
84	MNAME?	recalls name to Alpha
85	END	

Remarks.- The program is relatively short but hefty in data requirements: three auxiliary matrices are created and used during the calculations, meaning that the total numbers of registers needed (including the original matrix) is: 4 x dim (A)

Note also that the convergence is based on equal Frobenius norms of two consecutive iterations, and that the comparison is made using the full 9 decimal digits (see instruction "X#Y?"in line 75). A rounded comparison would result in shorter execution times, but it wouldn't be as accurate.

As usual, these routines will result in "ALPHA DATA" if the matrix is in LU decomposed form.

Square root of a Matrix { MSQRT }

In mathematics, the square root of a matrix extends the notion of square root from numbers to matrices. A matrix **B** is said to be a square root of **A** if the matrix product **BB** is equal to **A.**

Just as with the real numbers, a real matrix may fail to have a real square root, but have a square root with complex-valued entries. In general, a matrix can have many square roots, however, a positive-semidefinite matrix M (that satisfy that x * M x >=0 for all x in R^n) has precisely one positive-semidefinite square root, which can be called its principal square root.

Computing the matrix square root in the SandMatrix uses a modification of the the Denman-Beavers iteration. Let Y0 = A and Z0 = I, where I is the n × n identity matrix. The iteration is defined by

$$Y_{k+1} = \tfrac{1}{2}(Y_k + Z_k^{-1}),$$
$$Z_{k+1} = \tfrac{1}{2}(Z_k + Y_k^{-1}).$$

Convergence is not guaranteed, even for matrices that do have square roots, but if the process converges, the matrix Yk converges quadratically to a square root $A^{1/2}$, while Zk converges to its inverse, $A^{-1/2}$

Contrary to the exponential and logarithm programs, the square root convergence is checked using the rounded values of the norms for two consecutive iterations. You can set FIX 9 for maximum accuracy (and longest run time – not a problem on V41 and on the 41CL of course).

Example 1. Find a square root of the 3rd. order Hilbert matrix:

A = [[1 1/2 1/3]
[1/2 1/3 1/4]
[1/3 1/4 1/5]]

We'll use IMR to input the element vaules (as **PMTM** is not really suitable for this example). Previously we need to create the matrix, as follows:

ALPHA, "HILB3", ALPHA
3.003, XEQ "**MATDIM**"

Once all elements are entered, we execute MSQRT, which shows the norms of the different iterations. Let's assume we set the calculator in FIX 9 for the maximum accuracy available; then the result matrix is as follows:

Final Frobenius norm = 1,238278374

Sqrt(**A**) = [[0,917390290 0,345469265 0,197600714]
[0,345469265 0,374984280 0,270871020]
[0,197600714 0,270871020 0,295943995]]

Squaring the result matrix again (you can use M^2 for that) we can check the accuracy:

[Sqr(**A**)]^2 = [[0,999999999 0,499999999 0,333333333]
[0,500000000 0,333333333 0,250000000]
[0,333333333 0,249999999 0,200000000]]

which isn't bad at all for a 33 years old calculator indeed…

Example 2.- Find a square root of the 4 x 4 matrix below, and check the accuracy by squaring it back.

$$A = \begin{vmatrix} 56 & 97 & 17 & 89 \\ 33 & -68 & -42 & 5 \\ -206 & -48 & -34 & -104 \\ -39 & 92 & 27 & 30 \end{vmatrix}$$

Using FIX 4 and **PMTM** for the data input (nice integer values), the result is as follows:

$$SQRT(A) = \begin{vmatrix} 8.0000 & 6.0000 & 1.0000 & 7.0000 \\ -7.0000 & -1.0001 & -8.0000 & 3.0000 \\ -8.0001 & 6.0000 & 8.0000 & -6.0000 \\ 6.0000 & 7.0000 & 7.0000 & 3.0000 \end{vmatrix}$$

which is exact to 4 decimal places save a couple of *ulps* here and there.

The program listing is shown below. Note the relatively short program, but here too the data requirements are equally hefty as three auxiliary matrices are required, for a total of 4 x dim(**A**) registres needed either in main or X-memory (including the original matrix).

LBL "MSQRT"	
DIM?	*get dimension*
I#J?	*is it square?*
-ADV MATRX	*no, show error*
CF 00	
FNRM	*initial norm*
STO 00	*store it in R00*
ASTO 01	*matrix name to R01*
RDN	*dimension to X-reg*
"P"	
MATDIM	*auxiliary matrix P*
"Q"	
MATDIM	*auxiliary matrix Q*
MIDN	
LBL 00	
"Q,#"	
MINV	
MAT=	*auxiliary matrix #*
CLA	
ARCL 01	
"|-,#,P"	
MAT+	
"P,X"	
2	
MAT/	
FNRM	*Frobenius norm*
VIEW X	*show progress*
X<> 00	*swao with old norm*
RCL 00	*recall new one again*
X=YR?	*are they equal>*
SF 00	*yes, flag this fact*
X=YR?	*are they equal>*
GTO 02	*yes, jump over*
CLA	*no, keep at it*
ARCL 01	
"|-,#"	*prepare Alpha string*
MINV	*invert matrix*
MAT=	*copy in auxiliary*
MINV	*undo the inversion*
"Q.#,Q"	
MINV	*invert auxiliary*
MAT+	*sum it to partial result*
"Q,X"	
2	
MAT/	*divide by scalar 2*
LBL 02	
"P,"	
ARCL 01	
MAT=	
FC? 00	*were norms equal?*
GTO 00	*no, next iteration*
PURFL	*purge P*
"Q"	
PURFL	*purge Q*
"#"	
PURFL	*purge #*
MNAME?	*matrix name to Alpha*
END	*done*

As usual, this routine will result in "ALPHA DATA" if the matrix is in LU decomposed form.

Matrix Integer Powers and Roots. { $\boxed{\text{M^2}}$, $\boxed{\text{MPWR}}$, $\boxed{\text{M^1/X}}$ }

This application will be dealt with using a relatively brute force approach, in that the powers will be computed by successive application of the matrix multiplication; therefore the restriction to integer powers.

$\boxed{\text{MPWR}}$ calculates the general case n, whilst $\boxed{\text{M^2}}$ is used to square a matrix (i.e. n=2). The first requires the matrix name in Alpha and the exponent in the X-register, whereas for the second only the matrix name in Alpha is needed.

The exponent may also be a negative integer. For that case the inverse matrix is calculated first, and the positive integer power is used for it. Lastly, for n=0 the result is the identity matrix of course.

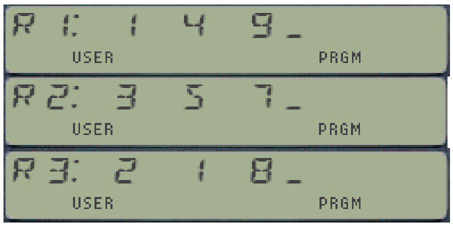
A feeble attempt is also made for the integer roots calculation: the function $\boxed{\text{M^1/X}}$ will attempt to calculate the x-th. root of a matrix using the general expression:

$[A]^{1/x} = \exp[1/x \,.\, \mathrm{Ln}(A)],$ which is only valid when $\mathrm{abs}(||A-I||) < 1$

Despite the inherent limitations of these programs they are interesting examples of extension of the "native" matrix function set, and therefore their inclusion in the SandMatrix.

Example1. Calculate the 7-th. power of the matrix below:

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

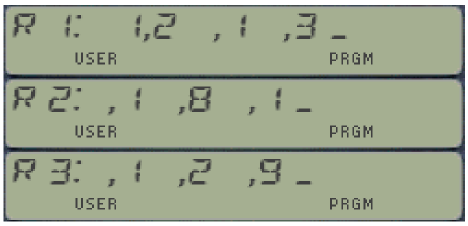


Type XEQ "**MPWR**", and the result is:

$$\mathbf{A}^7 = \begin{bmatrix} 7851276 & 8652584 & 31076204 \\ 8911228 & 9823060 & 35267932 \\ 5829472 & 6422156 & 23076808 \end{bmatrix}$$

Example 2. Calculate the 5th. root of matrix A below, then compare its 5th power to the original matrix.

$$\mathbf{A} = \begin{bmatrix} 1.2 & 0.1 & 0.3 \\ 0.1 & 0.8 & 0.1 \\ 0.1 & 0.2 & 0.9 \end{bmatrix}$$



The results are as follows:

$$\mathbf{A}^{1/5} = \begin{bmatrix} 1{,}034632528 & 0{,}015156701 & 0{,}057916477 \\ 0{,}019601835 & 0{,}953558110 & 0{,}020490861 \\ 0{,}017823781 & 0{,}045426856 & 0{,}974937998 \end{bmatrix}$$

$$[\mathbf{A}^{1/5}]^5 = \begin{bmatrix} 1{,}199999994 & 0{,}100000000 & 0{,}300000000 \\ 0{,}100000000 & 0{,}800000000 & 0{,}100000000 \\ 0{,}100000000 & 0{,}200000000 & 0{,}900000000 \end{bmatrix}$$

Program listings for **MPWR**, **M^2** and **M^1/X**.

<table>
<tr><td>1</td><td>LBL "MPWR"</td><td>***MNAME in Alpha***</td></tr>
<tr><td>2</td><td>DIM?</td><td>*get dimension*</td></tr>
<tr><td>3</td><td>I#J?</td><td>*square?*</td></tr>
<tr><td>4</td><td>-ADV MATRX</td><td>*yes, show error*</td></tr>
<tr><td>5</td><td>-CCD MATRX</td><td>*no, show "RUNNING…"*</td></tr>
<tr><td>6</td><td>X<>Y</td><td>*power index to X-reg*</td></tr>
<tr><td>7</td><td>INT</td><td>*make integer*</td></tr>
<tr><td>8</td><td>X#0?</td><td>*is it zero?*</td></tr>
<tr><td>9</td><td>GTO 01</td><td>*no, skip over*</td></tr>
<tr><td>10</td><td>MIDN</td><td>*yes, make identity*</td></tr>
<tr><td>11</td><td>RTN</td><td>*done.*</td></tr>
<tr><td>12</td><td>LBL 01</td><td></td></tr>
<tr><td>13</td><td>X<0?</td><td>*is it negative?*</td></tr>
<tr><td>14</td><td>MINV</td><td>*yes, invert matrix*</td></tr>
<tr><td>15</td><td>ABS</td><td></td></tr>
<tr><td>16</td><td>E</td><td></td></tr>
<tr><td>17</td><td>-</td><td>*n-1*</td></tr>
<tr><td>18</td><td>X=0?</td><td>*was n=1?*</td></tr>
<tr><td>19</td><td>RTN</td><td>*yes, we're done*</td></tr>
<tr><td>20</td><td>STO 00</td><td>*store in R00*</td></tr>
<tr><td>21</td><td>ASTO 01</td><td>*store Mname in R01*</td></tr>
<tr><td>22</td><td>*"|-,#"*</td><td></td></tr>
<tr><td>23</td><td>MAT=</td><td>*copy to aux matrix #*</td></tr>
<tr><td>24</td><td>DIM?</td><td>*get dimansion*</td></tr>
<tr><td>25</td><td>"P"</td><td></td></tr>
<tr><td>26</td><td>MATDIM</td><td>*auxiliary matrix P*</td></tr>
<tr><td>27</td><td>LBL 00</td><td>*prepare alpha string*</td></tr>
<tr><td>28</td><td>*"#,"*</td><td>*"#,"*</td></tr>
<tr><td>29</td><td>ARCL 01</td><td>*"#,MNAME"*</td></tr>
<tr><td>30</td><td>*"|-,P"*</td><td>*"#,MNAME,P"*</td></tr>
<tr><td>31</td><td>M*M</td><td>*matrix product*</td></tr>
<tr><td>32</td><td>VIEW 00</td><td>*show current index*</td></tr>
<tr><td>33</td><td>*"P,#"*</td><td></td></tr>
<tr><td>34</td><td>CLST</td><td></td></tr>
<tr><td>35</td><td>MMOVE</td><td>*copy result to #*</td></tr>
<tr><td>36</td><td>DSE 00</td><td>*decrement index*</td></tr>
<tr><td>37</td><td>GTO 00</td><td>*loop back if not ready*</td></tr>
<tr><td>38</td><td>*"#,"*</td><td>*"#,"*</td></tr>
<tr><td>39</td><td>ARCL 01</td><td>*"#,MNAME"*</td></tr>
<tr><td>40</td><td>MAT=</td><td>*copy result to #*</td></tr>
<tr><td>41</td><td>PURFL</td><td>*purge #*</td></tr>
<tr><td>42</td><td>*"P"*</td><td></td></tr>
<tr><td>43</td><td>PURFL</td><td>*purge P*</td></tr>
<tr><td>44</td><td>MNAME?</td><td>*recal MNAME to Alpha*</td></tr>
<tr><td>45</td><td>END</td><td>*done.*</td></tr>
</table>

<table>
<tr><td>1</td><td>LBL "M^2"</td><td>***MNAME in Alpha***</td></tr>
<tr><td>2</td><td>DIM?</td><td>*get dimension*</td></tr>
<tr><td>3</td><td>I#J?</td><td>*is it square?*</td></tr>
<tr><td>4</td><td>-ADV MATRX</td><td>*yes, error message*</td></tr>
<tr><td>5</td><td>-CCD MATRX</td><td>*no, show 'Running…"*</td></tr>
<tr><td>6</td><td>ASTO L</td><td></td></tr>
<tr><td>7</td><td>*"|-,"*</td><td></td></tr>
<tr><td>8</td><td>ARCL L</td><td></td></tr>
<tr><td>9</td><td>*"|-,P"*</td><td>*"M,M,P"*</td></tr>
<tr><td>10</td><td>ASWAP</td><td>*"M,P,M""*</td></tr>
<tr><td>11</td><td>ASWAP</td><td>*"P,M,M"*</td></tr>
<tr><td>12</td><td>MATDIM</td><td>*auxiliary P*</td></tr>
<tr><td>13</td><td>ASWAP</td><td>*"M,M,P"*</td></tr>
<tr><td>14</td><td>M*M</td><td>*matrix product*</td></tr>
<tr><td>15</td><td>CLAC</td><td>*"M,M,"*</td></tr>
<tr><td>16</td><td>CLAC</td><td>*"M,"*</td></tr>
<tr><td>17</td><td>*"|-P"*</td><td>*"M,P"*</td></tr>
<tr><td>18</td><td>ASWAP</td><td>*"P,M"*</td></tr>
<tr><td>19</td><td>MAT=</td><td>*result to M*</td></tr>
<tr><td>20</td><td>PURFL</td><td>*purge P*</td></tr>
<tr><td>21</td><td>MNAME?</td><td>*MNAME to Alpha*</td></tr>
<tr><td>22</td><td>END</td><td>*done*</td></tr>
</table>

<table>
<tr><td>1</td><td>LBL "M^1/X"</td><td>***MNAME in Alpha***</td></tr>
<tr><td>2</td><td>1/X</td><td></td></tr>
<tr><td>3</td><td>STO 05</td><td>*store in R05*</td></tr>
<tr><td>4</td><td>MLN</td><td>*matrix logarithm*</td></tr>
<tr><td>5</td><td>RCL 05</td><td></td></tr>
<tr><td>6</td><td>*"|-,X"*</td><td>*prepare Alpha string*</td></tr>
<tr><td>7</td><td>ASWAP</td><td>*swap string*</td></tr>
<tr><td>8</td><td>MAT*</td><td>*scalar multiplication*</td></tr>
<tr><td>9</td><td>MNAME?</td><td>*recall MNAME*</td></tr>
<tr><td>10</td><td>MEXP</td><td>*exponential*</td></tr>
<tr><td>11</td><td>END</td><td>*done*</td></tr>
</table>

Remarks:- Both **MPWR** and **M^2** need one auxiliary matrix (**P**) to temporarily place the results of the matrix product – Additionally, **MPWR** needs a second auxiliary matrix (**#**) as well.

An alternative listing for **M^1/X** that includes a convergency check is shown in next page. Note how the calculations to check for the condition are a taxing step, in that it requires a scratch matrix to calculate its norm. On the positive side though, it'll spare us the wait for a non-convergent process that would take much longer until it's apparent so. So after some consideration the longer version is now in the module.

1	LBL "M^1/X"	MNAME in Alpha	19	E	
2	1/X	1/n	20	X>Y?	meets condition?
3	STO 05	save it in R05	21	GTO 00	yes, go on
4	DIM?	dimansion	22	"#"	no
5	I#J?	not square?	23	PURFL	get rid of scratch
6	-ADV MATRX	show error	24	"DIVRGNT"	
7	-CCD MATRX	show "RUNNING…"	25	PROMPT	show error message
8	ASTO 01	save MNAME in R01	26	LBL 00	
9			27	CLA	
10	"#"	scratch matrix	28	ARCL 01	MNAME to Alpha
11	MATDIM		29	MLN	matrix logarithm
12	MIDN	make it Identity	30	RCL 05	1/n
13	CLA		31	"|-,X"	prepare string
14	ARCL 01	MNAME to Alpha	32	ASWAP	
15	"|-,#,#"	prepare string	33	MAT*	element multiplication
16	MAT-	intermediate result	34	MNAME?	MNAME to Alpha
17	ASWAP		35	MEXP	exponential matrix
18	FRNM	get its norm	36	END	done

The scratch matrix is removed in case there is divergence, or reused to calculate the logarithm if not – thus at least it's not all a waste of time. If there is no convergence you may still go ahead and hit R/S after the error message to see how the precision factor keeps increasing until the "OUT OF RANGE" condition.

A general-purpose algorithm for the p-th. root.

The principal p-th root of a non-singular matrix A (det A # 0) may be computed by the algorithm:

$M_0 = A \qquad M_{k+1} = M_k [(2I + (p - 2) M_k) (I + (p - 1) M_k)^{-1}]^p$

$X_0 = I \qquad X_{k+1} = X_k (2I + (p - 2) M_k)^{-1} (I + (p - 1) M_k)$

where I is the Identity matrix

M_k tends to I as k tends to infinity
X_k tends to $A^{1/p}$ as k tends to infinity

The convergence is also quadratic if A has no negative real eigenvalue.

Lie Product of two Matrices. { $\boxed{\text{MLIE}}$ }

The lie product is defines as the resultimg matrix obtained from the difference between the right and left multiplications of the matrices or in equation form:

$$Lie(A,B) = - Lie(B,A) = AB - BA$$

Example.- Calculate the Lie product for matrices:

$$\mathbf{A} = \begin{bmatrix} 1 & 2 & 4 \\ 3 & 5 & 7 \\ 7 & 9 & 8 \end{bmatrix} \quad \text{and:} \quad \mathbf{B} = \begin{bmatrix} 1 & 4 & 1 \\ 5 & 9 & 2 \\ 6 & 5 & 3 \end{bmatrix}$$

The results are:

$\boxed{\text{ALPHA}}$, "A,B,C", $\boxed{\text{ALPHA}}$
XEQ "**MLIE**" -> $Lie(A,B) = \begin{bmatrix} 15 & 11 & -23 \\ 24 & 19 & -65 \\ 58 & 85 & -34 \end{bmatrix}$

$\boxed{\text{ALPHA}}$. "B,A,C", $\boxed{\text{ALPHA}}$
XEQ "**MLIE**" -> $Lie(B,A) = \begin{bmatrix} -15 & -11 & 23 \\ -24 & -19 & 65 \\ -58 & -85 & 34 \end{bmatrix}$

The program listing is shown on the left. Note the usage of auxiliary matrix # to temporatily hold the result of the two matrix products (always the same limitation imposed by **M*M**), and the extensive usage of the alpha string management functions, like **ASWAP** – necessary to deal with the three matrix names in the string.

In fact **SWAP** swaps the contents of the Alpha register around the ***first*** *comma* character encounterd,; which makes it so interesting in this case.

<table>
<tr><td>1</td><td>LBL "MLIE"</td><td>"OP1, OP2, RES" in Alpha</td><td>15</td><td>ARCL 00</td><td>"RES,#,RES"</td></tr>
<tr><td>2</td><td>XEQ 00</td><td>calculate [OP1][OP2]</td><td>16</td><td>MAT-</td><td></td></tr>
<tr><td>3</td><td>ST<>A</td><td>complete string to stack</td><td>17</td><td>"#"</td><td></td></tr>
<tr><td>4</td><td>MNAME?</td><td>RES</td><td>18</td><td>PURFL</td><td>purge #</td></tr>
<tr><td>5</td><td>ASTO 00</td><td></td><td>19</td><td>MNAME?</td><td></td></tr>
<tr><td>6</td><td>ST<>A</td><td>restores complete string</td><td>20</td><td>RTN</td><td>done</td></tr>
<tr><td>7</td><td>CLAC</td><td>"OP1,OP2,"</td><td>21</td><td>LBL 00</td><td></td></tr>
<tr><td>8</td><td>ABSP</td><td>"OP1,OP2"</td><td>22</td><td>DIM?</td><td>get dimension</td></tr>
<tr><td>9</td><td>ASWAP</td><td>"OP2,OP1"</td><td>23</td><td>ASWAP</td><td>"OP2, RES, OP1"</td></tr>
<tr><td>10</td><td>"|-,#"</td><td>"OP1,OP2,#"</td><td>24</td><td>ASWAP</td><td>"RES, OP1, OP2"</td></tr>
<tr><td>11</td><td>XEQ 00</td><td>calculate [OP2][OP1]</td><td>25</td><td>MATDIM</td><td>create RES</td></tr>
<tr><td>12</td><td>CLA</td><td></td><td>26</td><td>ASWAP</td><td>"OP1, OP2, RES"</td></tr>
<tr><td>13</td><td>ARCL 00</td><td>"RES"</td><td>27</td><td>M*M</td><td>matrix product</td></tr>
<tr><td>14</td><td>"|-,"#,"</td><td>"RES,#"</td><td>28</td><td>END</td><td>return</td></tr>
</table>

Matrix Trace and remaining functions. { MTRACE }

In linear algebra, the trace of an n-by-n square matrix A is defined to be the sum of the elements on the main diagonal (the diagonal from the upper left to the lower right) of A, i.e.,

$$\operatorname{tr}(A) = a_{11} + a_{22} + \cdots + a_{nn} = \sum_{i=1}^{n} a_{ii}$$

where aii represents the entry on the ith row and ith column of A. The trace of a matrix is the sum of the (complex) eigenvalues, and it is invariant with respect to a change of basis. Note that the trace is only defined for a square matrix (i.e., n ×n).

Some of the properties of the trace are quite interesting and useful for other calculations, like eigenvalues and even determinants. In particular one could use the relationship that defines the trace of a product of matrices:

$$\operatorname{tr}(X^{\mathrm{T}}Y) = \operatorname{tr}(XY^{\mathrm{T}}) = \operatorname{tr}(Y^{\mathrm{T}}X) = \operatorname{tr}(YX^{\mathrm{T}}) = \sum_{i,j} X_{i,j}Y_{i,j}.$$

If we use an identity matrix in place of Y on the equation above it's clear that: tr(A) = **SUM** {[A] o [i]}, where the "o" symbol denotes the Hadamard or entry-wise product - as obtained by **MAT***.

The program in the SandMath however uses a direct approach, summing the elements in the diagonal – it's faster and doesn't require any auxiliary matrix to hold intermediate results.

Eigenvalues relationships.

The trace of a matrix is intricately related to its eigenvalues. In contrast with the determinant (which is the product of its eigenvalues); if A is a square n-by-n matrix with real or complex entries and if λ1,...,λn are the eigenvalues of A (listed according to their algebraic multiplicities), then

$$\operatorname{tr}(A) = \sum_i \lambda_i. \qquad \det(A) = \prod_i \lambda_i.$$

Another powerful property relates the trace to the determinant of the exponential of a matrix, as follows: (Jacobi's formula):

$$\det(e^A) = e^{\operatorname{tr}(A)} .$$

MTRACE uses a single-element approach, basically adding all the elements in the principal diagonal. For small to mid-size matrices this is faster than a block-approach, redimensioning and transposing the matrix such as the one sketched below (coutesy of Thomas Klemm):

```
          DIM(n, n+1)       A^T         DIM(1, n)                  DET

|* . .|        |* . . .|        |* * *|                  |1|
|. * .|   ->   |* . . .|   ->   |. . 0|   ->   |* * *| x |1|   ->   |*|   ->   *
|. . *|        |* 0 0 0|        |. . 0|                  |1|
                                |. . 0|
```

Here's the sweet and short SandMatrix program listing, compared side-to-side to a block-approach alternative implementation – which also requires a scratch matrix if one wishes to keep the original matrix unchanged, as well as some additional steps for Alpha housekeeping.

Note how the alternative approach function **SUM** is used, whch removes the need to calculate the determinant in the last step of the sketch. Regardless, it's bigger and takes longer excution time, even without the test for square matrix condition.

1	LBL "MTRACE"	*MNAME in Alpha*	1	LBL "TRACE2"	
2	DIM?		2	"|-,#"	*prepare Alpha string*
3	I#J?	*square?*	3	MAT=	*make scratch*
4	-ADV MATRX	*no, show error*	4	ASWAP	*place in hot spot*
5	0	*initial sum value*	5	DIM?	*gets its dimensions*
6	MSIJA	*sets pointer to 1:1*	6	E	
7	LBL 05		7	I<>J	*0,001*
8	MRR+	*recall element*	8	+	
9	+	*add to partial result*	9	MATDIM	*add one more column*
10	FC? 09	*end of row*	10	TRNPS	*transpose it*
11	I+	*no, next row*	11	INT	
12	FC? 10	*end of matrix?*	12	MATDIM	*make it a column matrix*
13	GTO 05	*no, next element*	13	SUM	*summ all elements*
14	END	*done*	14	PURFL	*purge scratch*
			15	ASWAP	*bring focus to original*
			16	CLAC	*alpha housekeeping*
			17	ABSP	*to erase all tracks*
			18	END	

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

The last function in this chapter 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 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).

1	LBL "R/aRR"	*MNAME in Alpha*	19	RDN	*discard product*
2	DIM?	*get dimansion*	20	FC? 09	*end of row?*
3	I#J?	*not square?*	21	GTO 00	*no, get next element*
4	-ADV MATRX	*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*			

Sum of Diagonal and Crossed Elements products. { MPDS , ΣIJJI }

Other two functions directly related to the eigenvalues are MDPS and ΣIJJI. They compute sums of pairs of element multiplication, either for those in the diagonal (aii * a kk); or for "crossed" (i.e. opposite) ones, (aij * aji), with i#j – excluding the diagonal.-2*1 – 4*2 + 3*0

Armed with these functions the characteristic polynomial of a 3 x 3 matrix can be expressed very succinctly – as we'll see in Chapter 4 of the manual.

Example. Calculate the trace and the sums of diagonal and crossed elements for the matrix below:

$$\begin{bmatrix} -2 & 2 & -4 \\ -1 & 1 & 3 \\ 2 & 0 & -1 \end{bmatrix}$$

Tr(A) = -2 + 1 – 1 = -2
MDPS = (-2*1) – (1*1) + (2*1) = -1
Σaij aji = -2 * 1 – 4 * 2 + 3* 0 = -10

Program listings – easy does it, element-wise.

1	LBL "ΣIJJI"	MNAME in Alpha	1	LBL "MDPS"	MNAME in Alpha
2	DIM?	get dimension	2	DIM?	get dimansion
3	I#J?	not square?	3	I#J?	not square?
4	-ADV MATRX	error message	4	-AVD MATRX	show error
5	INT	n	5	CF 00	default case
6	E		6	3	
7	-	n-1	7	X<=Y?	is i >= 3?
8	E3/E+	1,00(n-1)	8	SF 00	flag case
9	CLA		9	0	initial sum
10	STO M		10	MSIJA	set pointer to 1:1
11	LBL 00		11	LBL 06	
12	RCL M	k,00(n-1)	12	MRR+	recall element
13	E		13	FS? 09	end of row?
14	E3/E+	1,001	14	GTO 00	yes, juom out
15	+	(k+1),00n	15	I+	no, increase row
16	STO N		16	MR	recall element
17	LBL 01		17	*	multiply
18	RCL M	k,00(n-1)	18	+	add to partial sum
19	INT	k	19	FC? 10	end of matrix?
20	RCL N	(k+1),00n	20	GTO 06	no, do next row
21	INT	k+1	21	LBL 00	
22	I<>J	does E3/ for integers	22	FC? 00	order >3?
23	+	(k+1),00(n+k+1)	23	RDN	yes, get result to X-reg
24	MSIJ	sel pointer	24	FC?C 00	order >3?
25	MR	recall element	25	RTN	yes, done.
26	X<>Y		26	0	
27	I<>J	does E3/ for integers	27	MSIJ	set pointer to 1:1
28	MSIJ	set pointer	28	RDN	ann to X-reg
29	RDN		29	MR	aoo
30	MR	recall element	30	*	a00 * ann
31	*	multiply them	31	+	add to the sum
32	ST+ O	add to partial sum	32	END	done
33	ISG N	increase row			
34	GTO 01	next element in row			
35	ISG M	increase colum			
36	GTO 00	next colum			
37	RCL O	partial sum to X-reg			
38	MNAME?	recall mname to Alpha			
39	END	done			

Appendix.- Square root of a 2x2 Matrix.

A square root of a 2x2 matrix **M** is another 2x2 matrix **R** such that **M = R**^2, where **R**^2 stands for the matrix product of **R** with itself. In many cases, such a matrix R can be obtained by an explicit formula. Let

$$M = \begin{pmatrix} A & B \\ C & D \end{pmatrix}$$

where A, B, C, and D may be real or complex numbers. Furthermore, let τ = A + D be the trace of **M**, and δ = (AD – BC) be its determinant. Let s be such that s^2 = δ, and t be such that t^2 = τ + 2s. That is,

$$s = \pm\sqrt{\delta} \qquad t = \pm\sqrt{\tau + 2s}$$

Then, if t ≠ 0, a square root of **M** is:

$$R = \frac{1}{t}\begin{pmatrix} A+s & B \\ C & D+s \end{pmatrix}$$

1	LBL "SQRT2"	*MNAME in Alpha*	16	"|-,"	*prepare string*
2	"|-,#"	*Prepare Alpha string*	17	ARCL T	*"M,#,M,#"*
3	MAT=	*create scratch*	18	ST+ X	*2s*
4	ASWAP	*bring to hot spot*	19	MTRACE	*tr*
5	MDET	*determinant*	20	R^	*get 2s to X-reg*
6	ABS	*asolute value*	21	+	*tr + 2s*
7	SQRT	*s*	22	SQRT	*t*
8	MIDN		23	MAT+	*[A] = [A] + s[I]*
9	R^	*get s to X-reg*	24	",X,"	
10	ASWAP	*"M,#"*	25	MAT/	*[A] = [A] / t*
11	ASTO T	*save MNAME in T*	26	"#"	
12	"X,,"		27	PURFL	*get rid of scratch*
13	MAT*	*# = s #*	28	MNAME?	*MNAME to Alpha*
14	CLA		29	END	*done*
15	ARCL T	*recall MNAME*			

There it is, directly without doing any iterations or finding inverses. Your assignment now is to write a short program to calculate the square root of a 2x2 matrix applying the formula above.- Go ahead and try your hand at it … or cheat and look below.-

Note,- Not as trivial as you may think because the LU decomposition performing the determinant will conflict with other functions needed. Therefore one scratch matrix should be used here as well.

Example: calculate one square root of the matrix given below, and compare its square power to it.

$$A = \begin{pmatrix} 8 & -2 \\ 6 & 1 \end{pmatrix} = \begin{pmatrix} \dfrac{8 \pm 2\sqrt{5}}{2 \pm \sqrt{5}} & \dfrac{-2}{2 \pm \sqrt{5}} \\ \dfrac{6}{2 \pm \sqrt{5}} & \dfrac{1 \pm 2\sqrt{5}}{2 \pm \sqrt{5}} \end{pmatrix}.$$

This concludes the core matrix sections; it's time now to embark into the fascinating journey of characteristic polynomials and eigenvalues, as a prelude to the advanced polynomial chapter.

4. Polynomials and Linear Algebra

4.1. ADV POLYN	

Linear algebra is the branch of mathematics concerning vector spaces, as well as linear mappings between such spaces. Such an investigation is initially motivated by a system of linear equations in several unknowns. Such equations are naturally represented using the formalism of matrices and vectors.

	Function	Description	Input / Output
1	**CHRPOL**	Characteristic Polynomial	Under prgm control
2	**EIGEN**	Eigen Values by SOLVE	Under prgm control
3	**#EV**	Subroutine for EIGEN	Under prgm control
4	**EV3**	Eigen values 3x3	Matrix in X-Mem
5	**EV3X3**	Eigen values 3x3	Prompts Matrix Elements
6	**JACOBI**	Symmetrical Eigenvalues	Under prgm control

4.1. Eigenvectors and Eigenvaules.

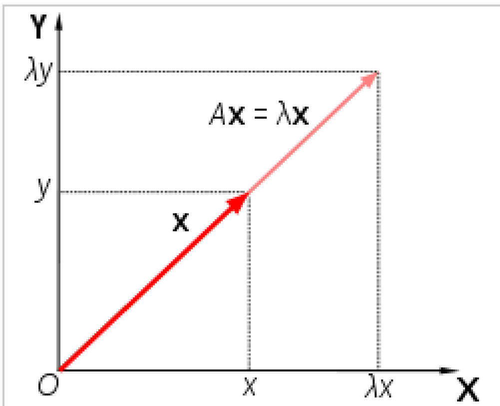
An eigenvector of a square matrix **A** is a non-zero vector ***v*** that, when the matrix is multiplied by ***v***, yields a constant multiple of v, the multiplier being commonly denoted by λ. That is:

$$Av = \lambda v$$

The number λ lis called the eigenvalue of **A** corresponding to ***v***.

In analytic geometry, for example, a three-element vector may be seen as an arrow in three-dimensional space starting at the origin. In that case, an eigenvector of a 3×3 matrix v is an arrow whose direction is either preserved or exactly reversed after multiplication by A.

The corresponding eigenvalue determines how the length of the arrow is changed by the operation, and whether its direction is reversed or not, determined by whether the eigenvalue is negative or positive.



A vector with three elements may represent a point in three-dimensional space, relative to some Cartesian coordinate system. It helps to think of such a vector as the tip of an arrow whose tail is at the origin of the coordinate system. In this case, the condition "u is parallel to v" means that the two arrows lie on the same straight line, and may differ only in length and direction along that line.

If we multiply any square matrix A with n rows and n columns by such a vector v, the result will be another vector w = A v , also with n rows and one column. That is,

$$\begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix} \text{is mapped to} \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \end{bmatrix} = \begin{bmatrix} A_{1,1} & A_{1,2} & \ldots & A_{1,n} \\ A_{2,1} & A_{2,2} & \ldots & A_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ A_{n,1} & A_{n,2} & \ldots & A_{n,n} \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix}$$

where, for each index i,

$$w_i = A_{i,1}v_1 + A_{i,2}v_2 + \cdots + A_{i,n}v_n = \sum_{j=1}^{n} A_{i,j}v_j$$

In general, if v is not all zeros, the vectors v and A v will not be parallel. When they are parallel (that is, when there is some real number λ such that A v = λ v) we say that v is an eigenvector of A. In that case, the scale factor λ is said to be the eigenvalue corresponding to that eigenvector.

In particular, multiplication by a 3×3 matrix A may change both the direction and the magnitude of an arrow v in three-dimensional space. However, if v is an eigenvector of A with eigenvalue λ, the operation may only change its length, and either keep its direction or flip it (make the arrow point in the exact opposite direction). Specifically, the length of the arrow will increase if $|\lambda| > 1$, remain the same if $|\lambda| = 1$, and decrease it if $|\lambda| < 1$. Moreover, the direction will be precisely the same if $\lambda > 0$, and flipped if $\lambda < 0$. If $\lambda = 0$, then the length of the arrow becomes zero.

4.4.4. Eigenvalues and eigenvectors of matrices: Characteristic Polynomial.

The eigenvalue equation for a matrix A is

$$Av - \lambda v = 0,$$

which is equivalent to

$$(A - \lambda I)v = 0,$$

where **I** is the n x n identity matrix. It is a fundamental result of linear algebra that an equation **M v = 0** has a non-zero solution ***v*** if, and only if, the determinant det(M) of the matrix **M** is zero. It follows that the eigenvalues of **A** are precisely the real numbers λ that satisfy the equation

$$\det(A - \lambda I) = 0$$

The left-hand side of this equation can be seen to be a polynomial function of the variable λ. The degree of this polynomial is n, the order of the matrix. Its coefficients depend on the entries of **A**, except that its term of degree n is always $(-1)^n \lambda^n$. This polynomial is called the characteristic polynomial of **A**; and the above equation is called the characteristic equation (or, less often, the secular equation) of **A**.

SOLVE-based Implementation. { EIGEN }

There are three Programs in the SandMatrix that calculate eigenvalues. The first one is aptly named **EIGEN**, and is a brute-force approach using the direct definition of the eigenvalue given above. What makes it interesting is the direct application of SOLVE (of **FROOT** in the SandMath) plus the combination of matrix functions to calculate the secular equation to solve for.

EIGEN can be used in manual mode (with guided prompts and data entry – or in a subroutine. In manual mode it creates a matrix named "**EV**" in X-mem. and will prompt for the elements data. In subroutine mode it'll take the matrix name from Alpha. You need to *set flag 06 for subroutine use,* or clear it for manual mode – this approach saves one FAT entry, although requires you to be aware of the rule.

The program checks that the matrix is square and not in LU-decomposed form – presenting error and warning messages respectively. For LU-decomposed matrices you can double-invert them "on the spot" (assuming they're invertible) and keep going just pressing R/S.

The selection of the interval [a,b] plays an important role in finding the solution – obviously the closer to the actual value the faster it'll find it. Remember also that the accuracy is determined by the display settings on the calculator, so FIX 9 will provide for both the most accurate and longest execution time.

Example. Find one eigenvalue for the matrix A below using the subroutine mode.

$$A = \begin{bmatrix} 3 & 1 & 5 \\ 3 & 3 & 1 \\ 4 & 6 & 4 \end{bmatrix}$$

Keystrokes	Display	Result
ALPHA, "**EV3**", ALPHA	X-reg contents	MNAME is in Alpha
3.003, XEQ"**MATDIM**"	3.003	Creates matrix in X-Mem
XEQ "**PMTM**"	"R1: _"	Prompts for the first row
3, ENTER^, 1, ENTER^, 5, R/S	"R2: _"	… second row
3, ENTER^, 3, ENTER^, 1, R/S	"R3: _"	… third row
4, ENTER^, 6, ENTER^, 4, R/S	6.0000	
SF 06	6.0000	Sets it in subroutine mode
XEQ "**EIGEN**"	"LO' V=?"	Prompts for lower bound
5, R/S	"HI' V=?"	Higher bound
15, R/S	flying goose…	FROOT is working on it
	"EV=10,00000"	ev found (in FIX 5).

The original matrix is not modified in any way, but note that an auxiliary matrix is created for the calculations. This scratch matrix "#" is not purged automatically from X-Mem, you'll have to do that after you're done calculating as many eigenvalues as you need.

Below is the program listing for **EIGEN**. Note how the equation to solve already requires an auxiliary FAT entry, **#EV** – since a global label is always needed by **FROOT**. (You can refer to the SandMath manual if you need to refresh your understanding of FROOT and FINTG)

1	LBL 02		26	"#"	scratch matrix
2	"LU FORM"	warning text	27	MATDIM	as identity one
3	AVIEW	display it	28	LBL 00	
4	MNAME?	MNAME back to Alpha	29	"LOW V'=?"	
5	STOP	***your chance to fix it***	30	PROMPT	prompt lower bound
6	GTO 01	try again	31	"HI V'=?"	
7	LBL "EIGEN"		32	PROMPT	prompt upper bound
8	ASTO 00	save MNAME in R00	33	-CCD MATRX	show "RUNNING…"
9	FS? 06	subroutine mode?	34	"#EV"	
10	GTO 01	yes, skip data entry	35	FROOT	Solve for Ev (!)
11	-SNDMATRX 4	prompts "ORDER=?"	36	TONE 4	found!
12	STOP		37	"EV="	
13	E		38	ARCL X	
14	E3/E+	1,001	39	PROMPT	display result
15	*	n,00n	40	GTO 00	next guess
16	"EV"	hard-coded name	41	LBL "#EV"	subroutine
17	MATDIM	create square matrix	42	"#"	
18	IMR	input elements	43	MIDN	make matrix identity
19	LBL 01		44	"X"	
20	ASTO 00		45	MAT*	multiply it by scalar guess
21	DIM?	get dimension	46	"#,"	
22	I#J?	not square?	47	ARCL 00	prepare Aplha string
23	-ADV MATRX	show error	48	"|-,#"	
24	LU?	LU decomposed?	49	MAT-	calculate the eigen matrix
25	GTO 02	yes, warning loop	50	MDET	get its determinant
			51	END	return

EIGEN works for N-dimensional orders. In that regard its execution time (provided that a decent initial guess is given) compares favorably to that of **CHRPOL**, the other program that calculates eigenvalues. The difference of course is that **CHRPOL** obtains *all the eigen values* simultaneously, whilst **EIGEN** does it one at a time.

When compared to other approaches, the program listed above is *almost minimalistic* – that's its real benefit and the reason that justifies its inclusion in the SandMatrix module. However relying on **FROOT** is clearly not a robust approach to calculate eigenvalues - The calculation of the characteristic polynomial using dedicated methods is a necessity.

3-Dimensional case. { EV3X3 , EV3 }

Let's start with the particular case n = 3. In this scenario there are simple formulas to calculate the characteristic polynomial, which make the calculations simpler and faster. The formulas are derived from the properties of the characterictic polynomial. Let's enumerate the most important ones.

The polynomial pA(x) is monic (its leading coefficient is 1) and its degree is n. The most important fact about the characteristic polynomial was already mentioned in the motivational paragraph: the eigenvalues of A are precisely the roots of pA(x). The coefficients of the characteristic polynomial are all polynomial expressions in the entries of the matrix. In particular its constant coefficient pA(0) is det(−A) = (−1)^n det(A), and the coefficient of x^(n−1) is tr(−A) = −tr(A), where tr(A) is the matrix trace of **A**. For a 2×2 matrix A, the characteristic polynomial is therefore given by:

$$\det(A) - \operatorname{tr}(A)\lambda + \lambda^2,$$

For a 3×3 matrix, the formula specifies the characteristic polynomial to be

$$\det(A) - c_2\lambda + \operatorname{tr}(A)\lambda^2 - \lambda^3 \ .$$

where c2 is the sum of the principal minors of the matrix = $\frac{1}{2}\left((\operatorname{tr}A)^2 - \operatorname{tr}(A^2)\right)$

Given the above definitions it is clear now why functions **MDPS** and **ΣIJJI** will be helpful to obtain the coefficients of the characteristic polynomial for n=3. In effect, when using those functions the formulas change as follows: **c2** = (**MDPS - ΣIJJI**)

For the manual mode (not as subroutine), a choice is offered to see the coefficients of the polynomial before calculating its roots (i.e. the eigenvalues).

, which will only take [Y] / [N] as valid inputs.

Example 1. Calculate the eigenvalues for **A**, with aij = ij

Solution: pA(x) = 75,349 x3 - 66 x2 -60 x =0

x1 = 66,890
x2 = -0,897
x3 = 2,24000E-9

Example 2. Calculate the eigenvaules for **A**, with aij = 1,2,3…9

Solution: pA(x) = 0,076 x3 -15 x2 -18 x = 0

x1= 16,117
x2 = -1,117
x3 = 2,89100E-9

It is therefore a relatively easy exercise to write a program to deal with this case, as shown in the program listing in next page.

1	LBL "EV3X3"		36	+	
2	CF 06	*clear subroutine flag*	37	X#Y?	
3	"EV"		38	GTO 01	*wrong key*
4	E		39	3	*choice accepted*
5	E3/E+	*1,001*	40	LBL 05	*coefficientes loop*
6	3		41	"b("	
7	*	*3,003*	42	AIP	
8	MATDIM		43	"|-)="	
9	IMR	*enter elements*	44	ARCL IND X	
10	GTO 06		45	PROMPT	
11	LBL "3EV"		46	E	
12	SF 06	*set subroutine flag*	47	-	
13	LBL 06		48	X<0?	*last one?*
14	-CCD MATRX	*show "RUNNING..."*	49	GTO 11	*yes, jump over*
15	MTRACE	*calculates tr(A)*	50	GTO 05	*no, get next one*
16	CHS	*change sign*	51	LBL 11	*proceed with roots*
17	STO 02	*save it in R02*	52	E	*fill stack with coeffs*
18	MDPS	*get the sum of minors*	53	RCL 02	
19	STO 01	*as a combination*	54	RCL 01	
20	ΣIJJI	*of functions into R01*	55	RCL 00	
21	ST- 01		56	CROOT	*calculate roots*
22	MDET	*calculate determinant*	57	FS?C 06	*was subroutine?*
23	CHS	*change sign*	58	RTN	*yes, end*
24	STO 00		59	"X="	*show results*
25	FS? 06	*subroutine mode?*	60	ARCL Z	*always a real one*
26	GTO 11	*yes, skip prompting*	61	PROMPT	
27	CF 21		62	FC? 43	*complex?*
28	"CFS? Y/N"	*offer choice*	63	GTO 01	*no, skip prompting*
29	AVIEW		64	X<> Z	*yes, clear Z*
30	LBL 01	*decode the Y/N input*	65	CLX	
31	GETKEY		66	X<> Z	
32	41		67	LBL 01	
33	X=Y?		68	QROUT	*show other two roots*
34	GTO 11	*choice rejected*	69	END	*done*
35	30				

Program remarks.-

Note that in manual mode **EV3X3** creates a matrix named "**EV**", but that the subroutine will work with *any* 3x3 matrix which name is in Alpha. This is compatible with **EIGEN** in its subroutine mode as well.

The roots are obtained using the SandMath function **CROOT**, an all-MCODE implementation of the Cardano-Vieta formulas. Function **QROUT** is also used to display them.

General case: N-dimensional general matrix. { CHRPOL }

The original CHRPOL - as it appeared in previous versions of the SandMatrix - was written by Eugenio Úbeda (as published in the UPLE), and later on adapted to the SandMatrix. Note however that it didn't make use of any advanced Matrix function, thus was pretty much the same as its initial version. It was a user-friendly program; providing step-by-step guidance for the data entry and didn't require any set-up preparation (like creating matrices) prior to the execution.

In this version CHRPOL has been re-written from the ground up, really taking advantage of the powerful matrix function set. It is a much improved solution, about twice as fast and with half the (comparable) code - It however now requires you to first create the matrix and input its elements.

Algorithmically it still uses the same modification of the Leverrier-Faddeev method to determine the coefficients of the characteristic equation of the n x n matrix; which roots are the eigenvalues of the matrix. It also employes the matrix trace in the process.

The coefficients are calculated using the iterations:

$b1 = -tr(\mathbf{B}1)$, with $\mathbf{B}1$ = the original matrix, and
$bk = -tr(\mathbf{B}k) / k$, with $\mathbf{B}k = \mathbf{A}(\mathbf{B}k\text{-}1 + bk\text{-}1\,\mathbf{I})$, $k=2,\dots n$

The program works for orders n between 3 and 14. The case n=2 has a trivial solution [given by b2=1, b1= tr(A), and b0 = -det(A)] ; therefore doesn't need to be included.

Example. Obtain the characteristic polynomial for the matrix A given below:

```
        [[    1   -0.69    0.28 ]
A =      [-0.69       1    0.18 ]
         [ 0.28    0.18       1 ]]
```

Keystrokes	Display	Result
ALPHA , "AA", ALPHA	current X-reg	Matrix name in Alpha
3.003 , XEQ "**MATDIM**"	3.003	Creates matrix in X-Mem
XEQ "**IMR**"	"a1,1= ?"	Prompts for data, also
1, R/S	"a1,2= ?"	showing current values
0.69, CHS, R/S	"a1,3= ?"	
0.28, R/S	"a2,1"= ?"	
0.69, CHS, R/S	"a2,2= ?"	
1, R/S	"a2,3= ?"	
0.18, R/S	"a3,1= ?"	
0.28, R/S	"a3,2= ?"	
0.18, R/S	"a3,3= ?"	Last element
1, R/S	1.000	
XEQ "**CHRPOL**"	"RUNNING…"	scrolls in the display, then
	"Σ(aK*X^K)"	Reminder of convention
(*) set F21	"a3=1,000000"	Coefficient of x^3
if you want AVIEW	"a2=-3.000000"	Coefficient of x^2
to stop each time	"a1=2.413100"	Coefficient of x
	"a0=-0.343548"	First coef (independent term).
	"RUNNING…"	Scrolls in the display, then
	"X=0,180390390"	First eigenvalue
R/S	"X=1,121568609"	Second eigenvalue
	"X=1,698238062"	Third and last.

See the program code below in its entire splendor – realizing that it may be the last program written using Advantage Matrix functions…

Remarks: Two auxiliary matrices are used, but the original matrix is left unaltered. The first part of the program (up to line 60) calculates the coefficients of the characteristic polynomial – and displays them for informational purposes. It then transfers the execution to the root finder routines. Note that for cases n=3 and n=4 we take advantage of the dedicated functions **CROOT** (in the SandMath) and **QUART**, which result in much faster execution than the general case using **RTSN** .

1	LBL "CHRPOL"	*MNAME in Alpha*	53	E3/E+	*1.001*
2	DIM?	*n,00n*	54	+	*1.00(n+1) - cnt'l word*
3	I#J?		55	*"#"*	
4	-ADV MATRX		56	PURFL	
5	ASTO 01	*MNAME*	57	*"P"*	
6	-CCD MATRIX	*shows 'RUNNING…"*	58	PURFL	
7	*"|-,P"*		59	PVIEW	*for information*
8	MAT=	*B = A*	60	-CCD MATRIX	*shows 'RUNNING…"*
9	ASWAP		61	PDEG	*new destination*
10	DIM?	*n,00n*	62	STO 00	*as expected by RTSN*
11	INT	*n*	63	*4*	
12	*E*		64	X>=Y?	n<=4?
13	+	*n+1*	65	*GTO 04*	*yes, particular case*
14	MDET	*independent term*	66	CLX	*no, general case*
15	STO IND Y	*stored in Rn+1*	67	*E*	
16	ASWAP		68	+	*n+1*
17	MAT=	*avoids LU issues*	69	*E6*	
18	DIM?		70	/	*0,000|00(n+1)*
19	*"#"*	*auxiliary array*	71	*3*	*build the "from,to"*
20	MATDIM		72	E3/E+	*1.003*
21	FRC	*0,00n*	73	+	*1.003|00(n+1)*
22	*2*		74	REGMOVE	*as expected by RTSN*
23	+	*2,00n*	75	RTSN	
24	STO 00		76	*GTO 00*	*go to EXIT*
25	CF 21	*not halting VIEW*	77	LBL 04	
26	LBL 00		78	X#Y?	*n#4?*
27	VIEW 00	*shows index*	79	*GTO 03*	
28	*"#"*		80	RCL 02	*a3*
29	MIDN	*[#] = [I]*	81	RCL 03	*a2*
30	*"P"*		82	RCL 04	*a1*
31	MTRACE	*tr (B)*	83	RCL 05	*a0*
32	RCL 00		84	QUART	
33	INT	*k+1*	85	*GTO 00*	*go to EXIT*
34	*E*		86	LBL 03	
35	-	*k*	87	RCL 01	*a3*
36	/		88	RCL 02	*a2*
37	CHS		89	RCL 03	*a1*
38	STO IND 00	*pk = -tr (B) / k*	90	RCL 04	*a0*
39	*"X,#,#"*		91	CROOT	
40	MAT*	*[#] = pk [I]*	92	*"X="*	
41	*"P,#,#"*		93	ARCL Z	
42	MAT+	*[#] = [B] + p[I]*	94	PROMPT	*real root*
43	CLA		95	FC? 43	*is RAD on?*
44	ARCL 01		96	*GTO 01*	*yes, complex roots*
45	*"|-,#,P"*		97	X<> Z	*no, real roots*
46	M*M	*B= A (B - p I)*	98	CLX	*so we clear Z*
47	ISG 00		99	X<> Z	
48	*GTO 00*		100	LBL 01	
49	DIM?	*n,00n*	101	QROUT	*output roots*
50	FRC	*0,00n*	102	LBL 00	
51	*E*		103	MNAME?	*bring MNAME back*
52	STO 01	*it's monic (!)*	104	END	*done*

Particular case: Symmetric Matrices { JACOBI }

For symmetric matrices the Jacobi algorithm provides a faster method. **JACOBI** was written by Valentín Albillo, and published in PPC TN, V1N3 (October 1980). As with CHRPOL, 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:

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

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

$$bip = aip \cos z - aiq \sin z$$
$$biq = aip \sin z + aiq \cos z$$
$$bik = aik\ ;\quad \text{where } i,k\ \#\ p,q$$

$$bpp = app \cos^2 z + aqq \sin^2 z - 2\ apq \sin z \cos z$$
$$bqq = app \sin^2 z + aqq \cos^2 z + 2\ apq\ \mathrm{sinz} \cos z$$
$$bpq = 0,\quad \text{and the remaining elements are symmetric.}$$

where: $\sin z = w / \mathrm{sqrt}(2(1+\mathrm{sqrt}(1-w\hat{}2)))$, and $\cos z = \mathrm{sqrt}\ (1-\sin^2 z)$
with: $L = -\ apq$, $M = (app-aqq) / 2$, and $w = L\ \mathrm{sign}(M) / \mathrm{sqrt}\ (M^2+L^2)$

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 usinf 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 minumim 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	"a1:1=?"	Data entry starts
25, R/S	"a1:2=?"	
41, CHS, R/S	"a1:3=?"	
10, R/S	"a1:4=?"	
6, CHS, R/S	"a2:2=?"	Note how the symmetric
68, R/S	"a2:3=?"	elements are skipped
17, CHS, R/S	"a2:4=?"	
10, R/S	"a3:3=?"	
5, R/S	"a3:4=?"	
3, CHS, R/S	"a4:4=?"	input the last element
2, R/S	"PREC.=?"	Asks for precision
5, R/S	"RUNNING…"	Scrolling on the display
	"X=0,03302"	
R/S	"X=98,52170"	After a while ~ 2.5m in normal 41
R/S	"X=1,18609"	the four ev's are displayed.
R/S	"X=0,25920"	

Example. Repeat the same case but using **CHRPOL**, to obtain the polynomial and its roots.

Keystrokes	**Display**	**Result**
ALPHA , "AA", ALPHA	current X-reg	Matrix name in Alpha
4.004, XEQ "**MATDIM**"	4.003	Creates mtrix in X-Mem
XEQ "**PMTM**"	"R1: _"	prompts for row-1
25, ENTER^, CHS, 41, ENTER^, 10, ENTER, CHS, 6, R/S		
	"R2: _"	prompte for row-2
CHS, 41, ENTER^, 68, ENTER^, CHS 17, ENTER^, 10, R/S		
	"R3: _"	prompts for row-3
10, ENTER^, CHS, 17, ENTER^, 5, ENTER^, CHS, 3, R/S		
	"R4: _"	prompts for row-4
CHS, 6, ENTER^, 10, ENTER^, CHS, 3, ENTER^, 2, R/S		
XEQ "**CHRPOL**"	"RUNNING…"	Scrolling on the display
R/S	"Σ(aK*X^K)"	Reminder of convention
	"a4=1"	Coefficient of x^4
	"a3=-100"	Coefficient of x^3
	"a2=146"	Coefficient of x^2
	"a1=-35"	Coefficient of x
	"a0=1,00000"	First coef. (independent term)
	"RUNNING…"	Scrolling on the display
	"X1=98,52170"	Frst root
R/S	"X2=1,18609"	Second root
R/S	"X3=0,25919"	Third root
R/S	"X4=0,03302"	Fourth and last root.

The solution is: **Chr(A) = x^4 -100 x^3 + 146 x^2 – 35 x +1**

4.2.- Managing Polynomials.

The remaing of this chapter is about polynomials. Let's first cover those functions used to manage the data entry and output for them, polynomial math and some handy utilities used in the other programs.

	Function	Description	Input / Output
7	**DTC**	Deleting Tiny Coefficients	Control word in X
8	**"P+P"**	Polynomial Sum	Driver for PSUM
9	**"P-P"**	Polynomial Subtraction	Driver for PSUM
10	**"P*P"**	Product of Polynomials	Driver for PPRD
11	**"P/P"**	Division of Polynomials	Driver for PDIV
12	**PCPY**	Polynomial Copy	Control word in X-reg, destination in Y
13	**PDIV**	Euclidean Division	Control words in Y- and X-regs
14	**PEDIT**	Edits Polynomial Coefficients	Control word in X-Reg
15	**PMTP**	Prompts for Coeffs in Alpha List	Control word in X-Reg
16	**PPRD**	Polynomial Multiplication	Control words in Y- and X-regs
17	**PSUM**	Polynomial Addition & Subtraction	Control words in Y- and X-regs
18	**PVAL**	Polynomial Evaluation	Control word in Y, argument in X
19	**PVIEW**	Views Polynomial Coefficients	Control word in X-Reg

4.2.1. Defining and Storing Polynomials.

A polynomial is an expression of the form

$$a_n x^n + a_{n-1} x^{n-1} + \cdots + a_2 x^2 + a_1 x + a_0,$$

where a(n)#0

Or, more concisely:

$$\sum_{i=0}^{n} a_i x^i$$

Polynomials can only be stored in main memory (ie. not as X-mem files), thus the way to handle them will be by a *control word* of the form **bbb.eee**, which denotes the beginning and end registers that hold the polynomial coefficients, a(i)

The coefficients are stored starting with the *highest order term first* (ie. x^n) in register bbb, and ending with the *zero-th term last,* stored in register eee. It follows that the degree of a polynomial n verifies: n = (eee –bbb).

For instance, the control word **1,007** represents a polynomial of degree 6, which coefficients are stored as follows: a(6) in R01, a(5) in R02, a(4) in R03, a(3) in R04, a(2) in R05, a(1) in R06 and a(0) in R07.

The Polynomial Editor. There are three functions available in the SandMatrix to enter and review polynomials in the calculator. The main one is **PEDIT**, which takes the input from the control word in the X-register and sequentially prompts for each coefficient value. The first thing it does is present a reminder of the convention used, relating the subindex to the power of the variable for each term:

A nice feature is that it'll check for available data registers to complete all the storage, re-adjusting the calculator SIZE if necessary. **PEDIT** does not use any data registers itself.

Note that **PEDIT** includes in the prompts the current value held in the corresponding data register, so you don't need to type a new one if it's already correct. Alternatively you can use **PVIEW** to review the coefficients without any editing capabilities. In this mode the prompts don't have the question mark at the end, which indicates the value cannot be changed from the program.

In edit mode

In review mode.

You can control wether PVIEW stops after each prompt or does the complete listing without stopping by setting or clearing the user flag 21. Note also that if the coefficient is an integer value it will not display the zeroes after the decimal point – in both editi and review modes.

A faster alternative for data entry is **PMPT** – the *polynomial prompt*. This one does for polynomials what **PMTM** did for matrices: the data entry is done as a list in Alpha, containing the values of all coefficients at once.

Obviously this is limited by the total length available in the Alpha register (24 characters), including the blank spaces that separate each entry, and the minus signs for negative values. The two leftmost characters in the prompt indicate the first data register used to sore the coefficients (not the row# as in the Matrix case). These characters are not part of the final list, and therefore aren't included in the total count.

Another restriction of **PMTP** is that values cannot be expressed in exponential form (using EEX), which key is ignored during the process. You *can* use negative and decimal values as the CHS and [,] (radix) keys are active. Obviously the back arrow key is always active to correct wrong entries.

1	**LBL "PEDIT"**		27	**"|-="**	
2	**SF 08**	*flags mode*	28	**RCL IND Y**	*append current value*
3	**ENTER^**	*copies cntl word to Y*	29	**FRC?**	*has fractional part?*
4	**I<>J**	*swaps bbb and eee*	30	**ARCL X**	*yes, append as is*
5	**E**		31	**INT?**	*integer*
6	**+**		32	**AINT**	*yes, append IP only*
7	**SIZE?**	*current size*	33	**FC? 00**	*editable?*
8	**X<>Y**		34	**AVIEW**	*no, show already*
9	**X>Y?**	*not enough?*	35	**FC? 08**	*editable?*
10	**PSIZE**	*adjust size*	36	**GTO 02**	*no, next coeff*
11	**RDN**		37	**LBL 00**	
12	**RDN**	*cntl word to X-reg*	38	**"|-?"**	*append "?"*
13	**GTO 00**	*skip over*	39	**CF 22**	*reset data entry flag*
14	**LBL "PVIEW"**		40	**PROMPT**	
15	**CF 00**	*flags mode*	41	**FC? 22**	*value entered?*
16	**LBL 00**		42	**GTO 02**	*no, next coeff*
17	**-ADV POLYN**	*shows convention*	43	**STO IND Z**	*yes, store it*
18	**PSE**		44	**RDN**	*discard entry*
19	**ENTER^**	*copies cntl word to Y*	45	**LBL 02**	
20	**PDEG**	*polyn degree*	46	**DSE X**	*decrement counter*
21	**X<>Y**	*cntl word to X-reg*	47	**NOP**	
22	**STO L**	*saves it in L*	48	**ISG Y**	*increment register*
23	**X<>Y**	*degree to X-reg*	49	**GTO 01**	*next register*
24	**LBL 01**		50	**LASTX**	*get control word*
25	**"a"**		51	**END**	*done*
26	**AIP**	*append index*			

4.2.2. Polynomial Arithmetic { PSUM , PPRD , PDIV }

The arithmetic functions provide convenient functionality for the basic operations: addition, subtraction, multiplication and eucliedean division. A distinction is made between the three base routines (**PSUM**, **PPRD**, and **PDIV** written by JM Baillard), and the four user-friendly drivers that automate the element data entry and work out all the details behind the scenes.

For the first group, beside the element data entry, the control words for each operand polynomial and the result are typically input in the X- , Y- and Z-registers of the stack. As follows:

Operation	Addition, Subtraction, Multiplication	Euclidean Division	Copy
Input	bbb.eee1 in Z bbb.eee2 in Y 1st. Reg of result in X	bbb.eee of dividend in Y bbb.eee of divisor in Y	bbb.eee of source in Y bbb or destination in X
Output	bbb.eee of result in X	bbb.eee of reminder in Y bbb.eee of quotient in X	bbb.eee or result in X

Because registers R00 to R03 are used internally, they cannot be used to hold the polynomial coefficients. (ie. all control words must start at bbb = 4 at least). Note also that none of the register ranges should overlap. In addition, for the Euclidean Division the original polynomials *are overwritten* with the results (quotient and reminder).

Let $a(x) = a_0.x^n+a_1.x^{n-1}+ ... + a_{n-1}.x+a_n$
and $b(x) = b_0.x^m+b_1.x^{m-1}+ ... + b_{m-1}.x+b_m$

then there are only 2 other polynomials q(x) and r(x) such that: a = b.q + r , with deg(r) < deg(b) .
Note that PDIV does not work if deg(a) < deg(b), but in this case q=0 and r=a.

Example 1.- Find the result of the polynomial product of a(x) * b(x), where:

$$a(x) = 2.x^5 + 5.x^4 - 21.x^3 + 23.x^2 + 3.x + 5 \quad \text{and} \quad b(x) = 2.x^2 - 3.x + 1$$

We'll use P*P for convenience. It'll automatically store the coefficients of the operand polynomial in registers {R04 to R09} and in registers {R10 to R12} respectively. The result polynomial will be stored starting with register R20, leaving the operand polynomials untouched.

The solution is: $p(x) = 4.x^7 + 4.x^6 - 55.x^5 + 114.x^4 - 84.x^3 + 24.x^2 - 12.x + 5$

Example 2.- Find the quotient and reminder for the polynomial division a(x) / b(x), where::

$$a(x) = 2.x^5 + 5.x^4 - 21.x^3 + 23.x^2 + 3.x + 5 \quad \text{and} \quad b(x) = 2.x^2 - 3.x + 1$$

We'll use P/P for convenience. It'll store the dividend coefficients in registers {R04 to R09} and the divisor's in registers {R10 to R12}. Note that in this case the coefficients are already there – as entered in the previous example, so you just have to press R/S during the process.

The solutions are displayed sequentially, starting with the quotient first. The indices convention message " Σ(aK*X^K)" is shown prior to the enumeration of each result polynomial. After completion, the control word for the reminder is left in X, and the control word for the quatient is saved in R00.

The solutions are: $q(x) = x^3 + 4.x^2 -5.x + 2$ and $r(x) = 14.x + 3$

Example 3.- Calculate the addition and subtraction of the polynomials a(x) and b(x) below:

$$a(x) = 2.x^3 + 4.x^2 + 5.x + 6 \quad \text{and} \quad b(x) = 2.x^3 - 3.x^2 + 7.x + 1$$

We'll use **P+P** and **P-P** for convenience. It'll automatically store the coefficients of the operand polynomials in registers {R04 to R07} and in registers {R08 to R11} respectively. The result polynomial will be stored starting with register R12, leaving the operand polynomials untouched. After completion, the control word for the result is left in X

The solutions are:

$$a(x) + b(x) = 4.x^3 + x^2 + 12.x + 7$$
$$a(x) - b(x) = 7.x^2 - 2.x + 5$$

Below you can see the program listing for the four arithmetic driver routines.

1	**LBL "P*P"**		32	**LBL 10**	
2	**CF 01**		33	*"N#1?"*	*order P1*
3	***GTO 00***		34	**PROMPT**	*n1*
4	**LBL "P/P"**		35	**4**	
5	**SF 01**		36	**+**	
6	**LBL 00**		37	**E3/E+**	*1,00(n+4)*
7	***XEQ 10***	*combined data entry*	38	**3**	
8	**FC? 01**	*product?*	39	**+**	*4,00(n+4)*
9	***GTO 00***	*yes, go there*	40	**STO 00**	
10	**RND**	*division*	41	**PEDIT**	
11	**PDIV**		42	***XEQ 05***	*adjust index*
12	**X<>Y**	*reminder cntl word*	43	**ENTER^**	*push stack*
13	**STO 00**	*store*	44	*"N#2?"*	*order P2*
14	**X<>Y**		45	**PROMPT**	*n2*
15	**PVIEW**	*show quotient*	46	**+**	*n2+eee1*
16	**X<> 00**		47	**I<>J**	*0,00(n2+eee1)*
17	***GTO 02***		48	**+**	*(eee1+1),00(eee1+n2)*
18	**LBL 00**	*multiplication*	49	**PEDIT**	
19	**PPRD**		50	**RCL 00**	*bbb.eee1*
20	***GTO 02***		51	**X<>Y**	*bbb.eee2*
21	**LBL "P+P"**		52	**LBL 05**	
22	**CF 01**		53	**ENTER^**	*bbb.eee2*
23	***GTO 01***		54	**I<>J**	*eee.bbb2*
24	**LBL "P-P"**		55	**INT**	*eee2*
25	**SF 01**		56	**E**	
26	**LBL 01**		57	**+**	*eee2+1*
27	***XEQ 10***	*combined data entry*	58	**END**	
28	**PSUM**				
29	**LBL 02**				
30	**PVIEW**	*show result (reminder)*			
31	**RTN**	*done*			

4.2.3. Deleting tiny Coefficients. { DTC }
Evaluating and Copying Polynomials. { PVAL , PCPY }

These three small routines were written by JM Baillard to perform the following housekeeping chores:

- Evaluate a polynomial value entered in the X-reg,
- Copy a polynomial from a source to a destination location, and
- Delete small coefficients (below 1E-7), wich typically appear due to rounding errors in the intermediate operations. This has a cumulative effect that can alter the final result if not corrected.

The evaluation leaves the result value in X. The other two functions return the destination control word to X upon completion. Below you can see the program listings for these; always a beauty to behold JM's mastery of the RPN stack.

1	LBL "PCPY"	
2	RCL Y	*bbb.eee1*
3	E3	
4	*	
5	INT	
6	I<>J	*does E3/ for integers*
7	SIGN	*puts bbb.eee in L*
8	RDN	
9	ENTER^	
10	ENTER^	
11	LBL 06	
12	CLX	
13	RCL IND L	
14	STO IND Y	
15	ISG Y	
16	CLX	
17	ISG L	
18	*GTO 06*	
19	CLX	
20	SIGN	
21	-	
22	I<>J	
23	+	
24	X<>Y	
25	FRC	
26	ISG X	
27	INT	
28	E5	
29	/	
30	+	
31	END	

1	LBL "PVAL"	***cnt'l word in X***
2	0	
3	LBL 14	
4	RCL Y	
5	*	
6	RCL IND Z	
7	*	
8	ISG Z	
9	GTO 14	
10	X<>Y	
11	SIGN	
12	RDN	
13	END	

1	LBL "DTC"	***cnt'l word in X***
2	LBL 05	
3	RCL IND X	
4	ABS	
5	E-7	*threshold value*
6	X<Y?	
7	*GTO 06*	
8	X<> Z	
9	ISG X	
10	*GTO 05*	
11	E	
12	ST- Y	*drecrease Y*
13	0	
14	STO IND Z	*overwrite w/ zero*
15	LBL 06	
16	X<> Z	*cnt'l word to X*
17	END	

When using PCPY be careful that the register ranges for both polynomials do not overlap.

4.3. Polynomial Root Finders.

Once upon a time there was a program called **POLYN** available in HP's infamous MATH PAC. That program was capable of calculating the roots of a polynomial up to degree *five*, which perhaps back then when it first came out could be regarded as a remarkable affair – but by today standards certainly isn't much to write home about.

	Function	Description	Input / Output
1	**QUART**	Solution of Quartic Equation	Polynomial coeffs in Memory
2	**PROOT**	Polynomial Roots	Prompts for all data
3	**RTSN**	Subroutine mode of PROOT	Polynomial coeffs in Memory
4	**BRSTW**	Quadratic Factors - Bairstow method	Cnt'l word in X-reg

The SandMatrix picks up where the SandMath left things off, providing functions to calculate the roots of the quadratic and cubic equations, ie. polynomials of degrees 2 and 3. The next step would then be a Quartic equation, or polynomial of degree 4.

4.3.1. Quartic Equation solutions. { QUART }

QUART solves the equation $x^4+a.x^3+b.x^2+c.x+d = 0$

If you have a polynomial not in monic form (which leading coefficient is not 1), simply divide all the equation by this coefficient. With this convention we can use the stack registers {T,Z,Y,X} to hold the coefficients a, b, c, and d – which provides a convenient method for data input.

The method used can be summarized as follows:

First, the term in x^3 is removed by a change of argument, leading to:

$$\mathbf{x^4+p.x^2+q.x+r = 0} \quad (E')$$

Then, the resolvant $z^3+p.z^2/2+(p^2-4r).z/16-q^2/64 = 0$ is solved by **CROOT,** and if we call z_1 , z_2 , and z_3 the 3 roots of this equation, the zeros of (E') are:

$$x = z_1^{1/2} \ \text{sign}(-q) +/- (z_2^{1/2} + z_3^{1/2});$$
$$x = -(z_1^{1/2}) \ \text{sign}(-q) +/- (z_2^{1/2} - z_3^{1/2})$$

Note that **QUART** uses R00 to R04 for scratch; therefore those registers cannot hold the polynomial.

The data output is done automatically by the program, presenting the roots as either real or complex conjugated. This is done using the status of flags 01 and 02 as appropriate – but the user needs not to concern him or herself with the decoding rules. The output uses function ZOUT from the SandMath, which uses "J" to denote the imaginary unit "i"

Example1: Solve $x^4 - 2.x^3 - 35.x^2 + 36.x + 180 = 0$

-2 ENTER^ , -35 ENTER^ 36 ENTER^, 180 , XEQ "**QUART**" >>>>

X1=6,000, X2=3,000

X3=-2,000 X4=-5,000

Example2: Solve $x^4 - 5.x^3 + 11.x^2 - 189.x + 522 = 0$

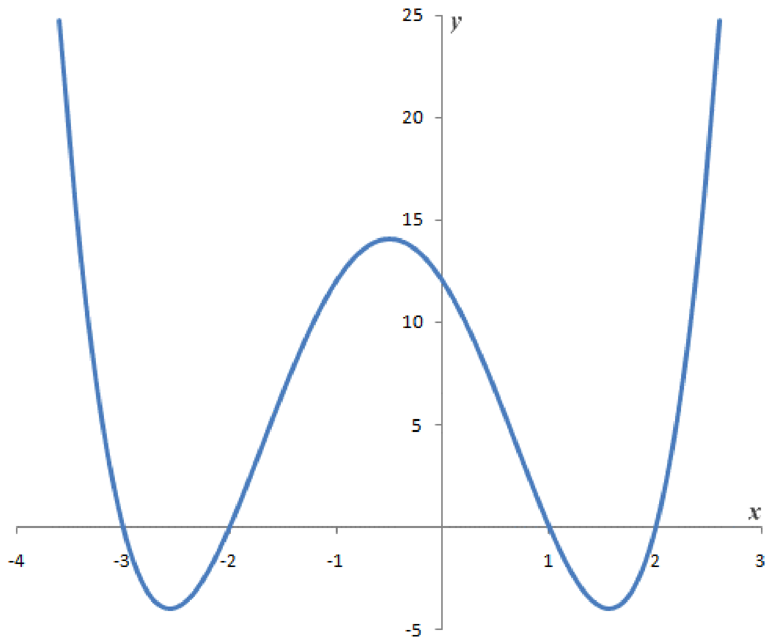
-5 ENTER^, 11 ENTER^, -189 ENTER^, 522 , XEQ "**QUART**" >>>>

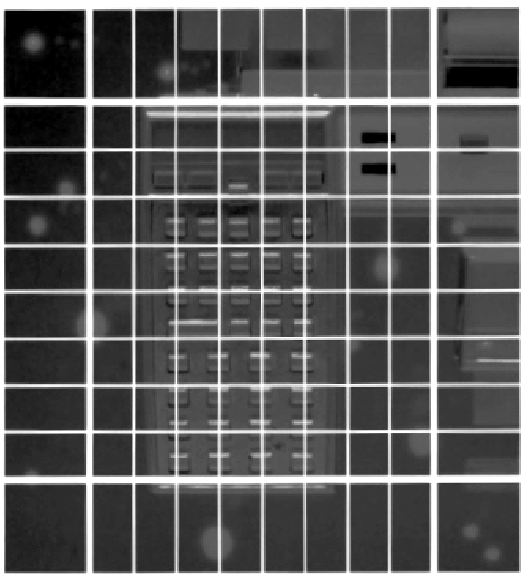
Z=-2+J5,000 (note how true integer values don't display zeros after the decimal point)
X3=3,000, X4=3,000

Example3: Solve $x^4 - 8.x^3 + 26.x^2 - 168.x + 1305 = 0$

-8 ENTER^ , 26 ENTER^ , -168 ENTER^ , 1305 , XEQ "**QUART**" >>>>

Z=-2+J5,000 (note how true integer values don't display zeros after the decimal point)
Z=6+J3,000





4.3.2. General case: degree N. (PROOT , RTSN , BAIRS }

Given a polynomial P,

$$P(z) = \sum_{i=0}^{n} a_i z^{n-i}, \quad a_0 = 1, \quad a_n \neq 0$$

This method is based on quadratic factorizations, that is one quotient polynomial of degree 2, plus a reminder polynomial of degree one - reducing the original degree by 2 and thereby changing the expression as follows:

P(z) = P"(z) Q(z) + R(z); with P"(z) = [Σ bi z^n-i] , i=2,1...(n-2)

This will then be repeated until the reduced polynomial P"(x) reaches degree one or two/.

Let Q(x) = x^2 + **p** x + **q**; and
R(x) = r x + s

Then the reduced polynomial coefficients are given by

bi = a(i-2) – p b(i-1) – q b(i-2) ; i = 2, 3, ..., (n+2) (1)

and we have the following expressions for the coefficients of the reminder:

r = b(n+1)
s = b(n+2) + p b(n+1) (2)

clearly with both r and s depending on the p,q values – formally expressed as: r=r(p,q) and s=s(p,q).

The problem is thus obtaining the coefficients p,q of such a quotient polynomial that ***would cancel the reminder*** (i.e. that make r=0 and s=0. This is accomplished by using an iterative approach, starting with some initial guesses for them (p0, q0), iterating until there is no change in two consecutive values,

r'(p,q) + r = 0; or: r'(p,q) = -r
s'(p,q) + s = 0; or: s'(p,q) = -s

Expressing it using their partial derivarives it results:

dp (δr/δp) + dq (δr/δq) = -r
dq (δs/δp) + dq (δs/δq) = -s

Using the relationships (1) above, we can formally obtain the partial derivatives using the coefficients of the original polynomial, ai. The problem will then be equivalent to solving a system of 2 linear equations with two unknowns, dp and dq.

From equation (1) above it follows:

δbi/δp = ci = -b(i-1) – p c(i-1) – q c(i-2); i = 2,3...,(n+2)
δbi/δq = c(i-1)

Making use of equation (2) to apply it for i=n we have as final expression

c(n+1) **dp** + cn **dq** = -b(n+1)
-q cn **dp** + [c(n+1) + p cn] **dq** = -[b(n+2) +p b(n+1)] (3)

Starting with (p0=0,5; q0=0,5) as initial guesses we'll obtain **dp** and **dq** for each pair of values (p,q). With them we adjust the previous guess, so that the new corrected values for p and q are

p' = p + **dp**
q' = q + **dq**

This will be repeated until the precision factor "ε" is smaller than the convergence criteria; The precision factor is calculated as follows:

ε = [abs(**dp**) + abs(**dq**)] / [abs(p) + abs (q)]

The program dimensions and populates matrices [**RS**] and [**CN**] to hold the current values of p,q and the coefficients Cn respectively:

- [**RS**] is the column matrix, of dimension (2x1).
- [**CN**] is the coefficients matrix, of dimension (2x2).

The linear system is solved as many times as iterations needed to establish the convergence. With each factorization the program obtains two roots. This is repeated for, until all roots have been found.

Program Details.

In manual (RUN) mode **PROOT** prompts first for the order n (ie. the degree) and for each of the coefficients sequentially. It then presents the option to store the roots into a matrix in X-Mem. To use it you just have to press "**Y**" at the prompt below:

All roots are stored in matrix [**ROOTS**], of dimension (n x 2) - with the first column holding the real parts and the second the imaginary parts of each root (assumed complex).

The global label **RTSN** is meant to be used in subroutines. It expects the degree stored in R00, and the coefficients stored in registers R03 until R(3+n). Registers R01 and R02 are used internally and cannot be used for your data. In subroutine mode the roots will always be stored in the matrix [**ROOTS**].

Example 1. Find the five roots of the polynomial below

$$P(x) = 2.x^5 + 7.x^4 + 20.x^3 + 81.x^2 + 190.x + 150$$

Keystrokes	**Display**	**Result**
XEQ "**PROOT**"	"ORDER=?	Prompts for the degree
5, R/S	"Σ(aK*X^K)"	Reminder of convention
	"a5= ?"	prompts for coeffs, showing current
2, R/S	"a4= ?"	
7, R/S	"a3= ?"	
20, R/S	"a2= ?"	
81, R/S	"a1= ?"	
190, R/S	"a0= ?	
150, R/S	"STO? Y/N"	prompts for storage option
"**Y**"	"RUNNING"	

At this point the different precision factors are shown, which shoud be decreasing as the iterations converge towards the solutions – and this repeated as many times at quadratic factors are needed.

The solutions are shown below (in FIX 5):

Z=-2,00000+J1,00000 and its conjugate (not shown)
Z=1,00000+J3,00000 and its conjugate (not shown)
Z=-1,50000

And the matrix [**ROOTS**] is left in X-Mem, with 5 rows and two columns, as follows:

$$[\mathbf{ROOTS}] = \begin{bmatrix} -2 & 1 \\ -2 & -1 \\ 1 & 3 \\ 1 & -3 \\ -1.5 & 0 \end{bmatrix}$$

To be sure it isn't the fastest method in town (typically 5-6 iterations are needed, each iteration takes a bout one full minute at normal speeds), but it's applicable to any degree and stores the results in a matrix – which makes it very useful as a general-purpose approach.

Bairstow Method.

A faster program is **BAIRS**, which also uses a factorization method but does not utilize any of the matrix functions. Therefore the solutions are just prompted to the display, but not saved into an X-Mem file. **BAIRS** expects the coefficients already stored in main memory, and the polynomial control word in X . Note that they will be overwritten during the execution of the program. It uses registers R00 to R08 internally, thus cannot be used to store your data.

For both programs the accuracy of the solutions (and therefore their run times) depends on the display settings.

BAIRS factorizes the polynomial
$p(x) = a_0.x^n+a_1.x^{n-1}+ \ldots + a_{n-1}.x+a_n$ into quadratic factors and solves $p(x) = 0$ (n >1)

If deg(p) is odd, we have $p(x) = (a_0.x+b).(x^2+u_1.x+v_1)........(x^2+u_m.x+v_m)$; with $m = (n-1)/2$
If deg(p) is even $p(x) = (a_0x^2+u_1.x+v_1)(x^2+u_2.x+v_2)......(x^2+u_m.x+v_m)$; with $m = n/2$

The coefficients u and v are found by the Newton method for solving 2 simultaneous equations. Then p is divided by $(x^2+u.x+v)$ and u & v are stored into R(ee-1) & Ree respectively . The process is repeated until all quadratic factors are found

Example 2. Solve $x^6 - 6.x^5 + 8.x^4 + 64.x^3 - 345.x^2 + 590.x - 312 = 0$

Using **PMTP** to store the coefficients beginning in R09, thus the control word is **9,015**

Keystrokes	**Display**	**Result**
9.015, XEQ "**PMTP**"	"R9: _"	
1, ENTER, CHS, 6, ENTER^,^8, ENTER^, 64, ENTER^, CHS, 345, ENTER^, 590, ENTER^, CHS, 312,		
R/S	9,015	
XEQ "**BAIRS**"	shows precisions factors...	

The solutions are: "Z=-4,000" and "Z=2,000"
"Z=2,000+J3,000" and conjugate (not shown)
"Z=1,000" and "Z=3,000"

4.4. Extended Polynomial Applications.

A few related topics - in that polynomials are involved - even if some programs don't make direct utilization of matrix functions. Here too the SandMatrix complements the functionallity included in the SadnMath. The table below summarizes them:

	Function	Description	Input / Output
1	**EQT**	Equation Display	Equation number in R00 0(1 to 15)
2	**POLINT**	Polynomial interpolation	Under program control
3a	**PRMF**	Prime Factors decomposition	Argument in X-reg
3b	**PF>X**	From prime factors to argument	Prime factors in matrix [PRMF]
3c	**TOTNT**	Euler's Totient function	Argument in X-reg
4	**POLFIT**	Polynomial Fitting	Under program control
5	**OPFIT**	Orthogonal Polynomial Fit	Under program Control
6a	**POLZER**	From Poles to Zeroes	Under program control
6b	**PFE**	Partial Fractions Expansion	Under program control

4.4.1. Displaying the Equations for Curve Fitting Programs { EQT }

As there was plenty of available space in the module, I decided to include this routine to complement the Curve Fitting program in the SandMath (**CURVE**). The routine EQT will write in Alpha the actual equation which reference number is in register R00, ranging from 0 to 15 as per the table below:

0. Linear
1. Reciprocal
2. Hyperbola
3. Reciprocal Hyperbola
4. Power
5. Modified Power
6. Root
7. Exponential
8. Logarithmic
9. Linear Hyperbola
10. 2nd. Order Hyperbolic
11. Parabola
12. Linear Exponential
13. Normal
14. Log Normal
15 Cauchy

Note that EQT does not perform any calculations, thus it's just an embellishing addition to CURVE.

The original listing was originally published in the AECROM manual, and it's reproduced here practically unaltered.

```
01•LBL "EQ"
02 XEQ IND 00
03 AVIEW
04 RTN
05 GTO "EQ"
06•LBL 00
07 "a+bX"
08 RTN
09•LBL 01
10 "1/(a+bX)"
11 RTN
12•LBL 02
13 "a+(b/X)"
14 RTN
15•LBL 03
16 "X/(aX+b)"
17 RTN
18•LBL 04
19 "aX↑b"
20 RTN
21•LBL 05
22 "ab↑X"
23 RTN
24•LBL 06
25 "ab↑(1/X)"
26 RTN
27•LBL 07
28 "ae↑(bX)"
29 RTN
30•LBL 08
31 "a+bLNX"
32 RTN
33•LBL 09
34 "a+bX+(c/X)"
35 RTN
36•LBL 10
37 "a+(b/X)+(c/X↑2)"
38 RTN
39•LBL 11
40 "a+bX+cX↑2"
41 RTN
42•LBL 12
43 "aX/bX"
44 RTN
45•LBL 13
46 "ae↑(((X-b)↑2)/c"
47 "⊢)"
48 RTN
49•LBL 14
50 "ae↑(((b-LNX)↑2)"
51 "⊢/c)"
52 RTN
53•LBL 15
54 "1/((a(X+b)↑2)+c"
55 "⊢)"
56 RTN
57 END
```

4.4.2. Polynomial interpolation. { POLINT }

The program **POLINT** follows the Aitken's interpolation method. It's an elegant simple implementation and a nice example of utilization of the capabilities of the platform. It was written by Ulrich K. Deiters, and it is posted at: http://www.hp41.org/LibView.cfm?Command=View&ItemID=600

The program performs polynomial interpolations of variable order on (xi, yi) data sets, with the order determined by the number of data pairs. It is applied as follows:

- You have a set of (xi, yi) data pairs. The xi are all different, and they need not be equidistant.

- You need to know the y value at the location x, which is not one of the xi.

- You start the program and enter x at the prompt.	XEQ "**POLINT**" x, R/S
- Then you enter the first data pair, preferably one which has an x_i close to x. The program returns y0.	x0, R/S y0, R/S
- You enter another data pair. The program returns the results of a linear interpolation.	R/S x1, R/S y1, R/S
- You enter another data pair. The program returns the results of a quadratic interpolation.	R/S x2, R/S y2, R/S
- You enter another data pair. The program returns the results of a cubic interpolation.	R/S x3, R/S y3, R/S

- ... and so on, until you exceed the SIZE of your calculator.

Going beyond the cubic interpolation is seldomly useful. High-order interpolations become increasingly sensitive to round-off errors and inaccuracies of the input data.

The number of data registers used depends on the order of the interpolation. An nth order interpolation (which uses n+1 pairs of data) occupies the registers R00 to R(2n+4), e.g., a cubic interpolation needs all registers up to R10.

If a printer is connected, the interpolation results are printed out, and the "empty" R/S entries are not required.

Example. Given the table below with a set of vapor pressure data for superheated water, what is the vapor pressure at 200 °C (= 473.15 K)?

T/K	380	400	450	480	500	530	560
p/MPa	0.12885	0.24577	0.93220	1.7905	2.6392	4.4569	7.1062

Here's the sequence followed to resolve it.

input	**display**	
XEQ "**INTPOL**"	X=?	
473.15, R/S	X0=?	
480 , R/S	Y0=?	
1.7905 , R/S	Y = 1.79050	
R/S	X1=?	
450 ,R/S	Y1=?	
0.9322, R/S	Y = 1.59452	linear interpolation
R/S	X2=?	
500, R/S	Y2=?	
2.6392, R/S	Y = 1.55067	quadratic interpolation
R/S	X3=?	
400 ,R/S	Y3=?	
0.24577, R/S	Y = 1.55453	cubic interpolation
R/S	X4=?	
530, R/S	Y4=?	
4.4569, R/S	Y = 1.55495	4th order

From this we conclude that 1.55 MPa is a reasonably good estimate; and that the linear interpolation was certainly not sufficient. Incidentally, the true value is 1.554950 MPa..

The program listing is shown below.

1	LBL "POLINT"	
2	FC? 55	
3	SF 21	
4	*"X=?"*	
5	PROMPT	x value of point
6	STO 00	
7	*3,05*	
8	STO 01	
9	LBL 01	
10	RCL 01	
11	INT	*k*
12	*E*	
13	-	*k-1*
14	E3/E+	*1,00(k-1)*
15	*3*	
16	+	*4,00(k-1)*
17	STO 02	
18	RCL 01	
19	INT	*k*
20	*3*	
21	-	*k-3*
22	*2*	
23	/	
24	*"X"*	
25	AIP	
26	*"|-=?"*	
27	PROMPT	*prompts for Xk*
28	RCL 00	
29	-	
30	STO IND 01	
31	ISG 01	
32	*"Y"*	
33	X<>Y	
34	AIP	
35	X<>Y	
36	*"|-=?"*	
37	PROMPT	*prompts for Yk*
38	DSE 02	
39	*GTO 02*	
40	LBL 03	
41	RCL IND 02	
42	*	
43	LASTX	
44	RCL Z	
45	-	
46	ISG 02	
47	RCL IND 02	
48	LASTX	
49	*	
50	ST- Z	
51	LASTX	
52	RDN	
53	RDN	
54	/	
55	LBL 02	
56	STO IND 01	
57	ISG 02	
58	*GTO 03*	
59	*"Y="*	
60	ARCL X	
61	AVIEW	
62	ISG 01	
63	*GTO 01*	*next order*
64	END	*done*

4.4.3. Prime Factors Decomposition { PRMF , PF>X , TOTNT }

This section describes the three functions provided in the SandMatrix related to Prime factorization.

	Function	Description	Input / Output
1	**PRMF**	Prime Factors (Matrix Form)	Argument in X-reg
2	**PF>X**	From Factors to Number	Prime factors in Matrix file
3	**TOTNT**	Euler's Totient function	Argument in X-reg

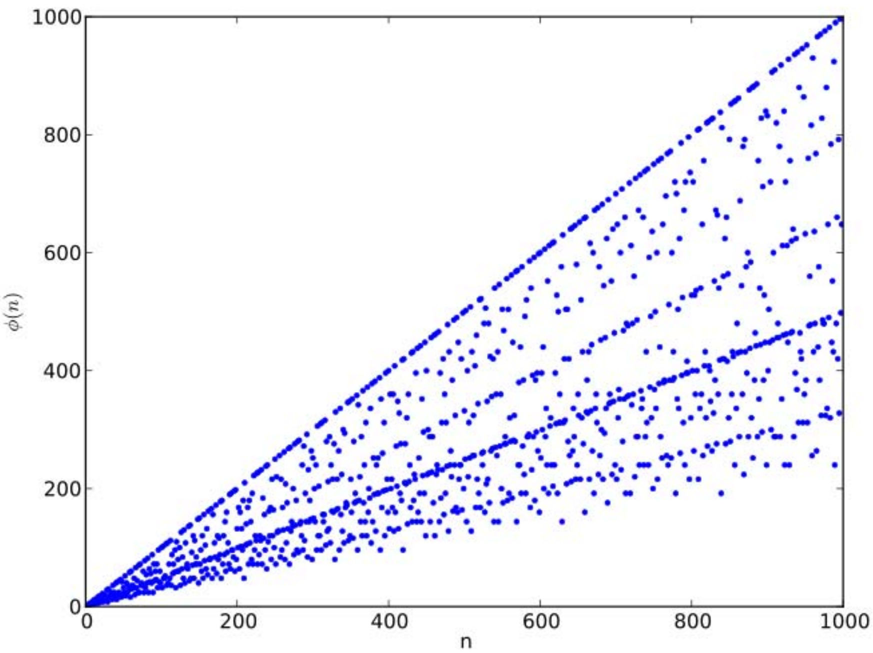
The first one **PRMFC** extends the basic prime factorization capability in the SandMath, **PFCT**. The difference is that whereas PFCT only uses the Alpha register to output the result (as Alpha string), here the prime factors and their multiplicities are also stored in a matrix in X-Mem - named [**PRFM**]. This ensures that no information will be lost (scrolled off the display if the length exceeds 24 char), and also provides a permanent storage of the results.

You can use **PF>X** to check the result: it re-builds the original argument from the values in the [**PRMF**] matrix, using the obvious relationship:

X = $\prod$ **PF(i) ^m(i) ;** for i = 1, 2… primes

Euler's Totient function.

In number theory, Euler's totient or phi function, φ(n) is an arithmetic function that counts the totatives of n, that is, the positive integers less than or equal to n that are relatively prime to n. The graphic below shows (well, sort of) the first thousand values of φ(n)



Examples. Calculate the prime factors and the totient for the following numbers:

n	PF	phi
1,477	7*211	1,260
819,735	3*5*7*37*211	362,880
123,456	2^6*3*643	41.088,000

The programs are listed below. There's no fancy algorithm for **TOTNT**, it just counts the number of prime factors after doing the decomposition as a preliminary step.

1	LBL "TOTNT"	***Euler's Totient Function***	55	*GTO 03*	skip if yes
2	SF 04	*flag case*	56	ST/ L	*divide number by PF*
3	*XEQ 10*	*get all Prime Factors*	57	LASTX	*Reduced number*
4	0		58	*GTO 00*	*loop back*
5	MSIJ	*sets pointer to 1:1*	59	LBL 03	*Store Exponent*
6	X<>Y	*argument to x*	60	RCL 00	recover PF
7	LBL 07		61	MSR+	store in matrix
8	MRC+	*get element*	62	*GTO 01*	*next factor*
9	1/X	*invert it*	63	LBL 02	*New PF found*
10	CHS	*sign change*	64	STO 01	*Store for comparisons*
11	*E*		65	RCL 00	previous exponent
12	+	*add 1*	66	MSR+	***Store Old PF Exponent***
13	*	*multiply*	67	RDN	
14	FC? 09	*end of row?*	68	ST/ L	*divide number by PF*
15	*GTO 07*	*loop back*	69	LASTX	*Reduced number*
16	CLD	*refesh display*	70	DIM?	
17	RTN	*done.*	71	X<> Z	*Bring the new PF back*
18	LBL "PRMF"	***Prime Factors***	72	MSR+	*store new PF*
19	CF 04	*flag case*	73	FS?C 00	*Was it Prime?*
20	LBL 10		74	*GTO 01*	*Bail Out, we're done*
21	*"PRMF"*		75	X<>Y	*Bring the number back*
22	2		76	*GTO 05*	*Start Over*
23	E3/E+	*1,002*	77	LBL "PF>X"	***Rebuild number***
24	MATDIM	*Create Matrix*	78	SF 04	*flag case*
25	CLX		79	*"PRMF"*	*matrix name*
26	MSIJA	*sets pointer to 1:1*	80	SF 10	*fake condition*
27	RDN	*argument to x*	81	LBL 01	*PF Completed*
28	CF 00	*default: not prime*	82	*E*	*1*
29	INT	*condition x*	83	FC? 10	*end of matrix?*
30	ABS	*to avoid errors*	84	MSR+	*store it as last exp.*
31	PRIME?	*is it prime?*	85	STO 00	initial value
32	SF 00	*FIRST PF found*	86	MSIJA	*sets pointer to 1:1*
33	MSR+	*Store this PF*	87	CLA	*Clean Slate*
34	X=1?	*is PF =1?*	88	LBL 06	*Rebuild the number*
35	*GTO 01*	*yes, leave the boat*	89	MRR+	*get prime factor*
36	FS?C 00	*Was it Prime?*	90	FC? 04	*if not totient case*
37	*GTO 01*	*if Prime, we're done*	91	AIP	*add it to Alpha*
38	STO 01	*Store PF for comparisons*	92	MRR+	*get multiplicity*
39	ST/ L	*divide number by PF*	93	FC? 04	*if not totient and/*
40	LASTX	*Reduced number*	94	X=1?	*or if it is one*
41	LBL 05		95	*GTO 04*	*skip adding to Alpha*
42	*E*	*reset counter*	96	*"|-^"*	*otherwise put symbol*
43	STO 00		97	AIP	*and add it to the string*
44	RDN		98	LBL 04	
45	LBL 00		99	Y^X	*PF^Exp*
46	RCL 01	recall PF	100	ST* 00	*Rebuilding the number*
47	X<>Y	Reduced number	101	FS?10	*End of Array?*
48	PRIME?	is it prime?	102	*GTO 04*	*yes, leave the boat*
49	SF 00	*PF found*	103	FC? 04	*if not totient case*
50	X#Y?	*Compare this and old PF's*	104	*"|-*"*	*append symbol*
51	*GTO 02*	*skip over if different*	105	*GTO 06*	*next PF*
52	ISG 00	*Same One*	106	LBL 04	
53	NOP	*Increase counter*	107	RCL 00	*final result*
54	FS?C 00	*Was it Prime?*	108	FC? 04	*if not totient case*
			109	AVIEW	*Show the construct*
			110	END	*done.*

4.4.4. Polynomial Fitting { POLFIT }

The next program is taken from Valent'in Albillo article "*Long Live the Advantage ROM*" - showcasing the matrix functions included in it. As one can expect from that reference, it's an excellent example and therefore more that worth including in the SadnMatrix.

The original article is partially reproduced below – it is so well described that I could not resist adding it practically verbatim.

POLFIT is a small, user-friendly, fully prompting 62-line program (124 bytes) written specifically to demonstrate the excellent matrix capabilities of the Advantage ROM. **POLFIT** can find the coefficients of a polynomial of degree N which exactly fits a given set of N+1 arbitrary data points (not necessarily equally spaced), where N is limited only by available memory.

Among the many functions we could fit to data, polynomials are by far the easiest to evaluate and manipulate numerically or symbolically, so our problem is:

Given a set of n+1 data points (x1, y1), …, (xn+1, yn+1), find an Nth-degree polynomial

y = P(x) = a1 + a2 x + a3 x2 + a4 x3 + ... + an+1 xn

which includes the (n+1) data points (x1, y1), (x2, y2), ..., (xn+1, yn+1). The coefficients (a1, ..., an+1) can be determined solving a system of (n+1) equations:

$$\begin{bmatrix} 1 & x_1 & x_1^2 & . & x_1^n \\ 1 & x_2 & x_2^2 & . & x_2^n \\ . & . & . & . & . \\ 1 & x_{n+1} & x_{n+1}^2 & . & x_{n+1}^n \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ . \\ a_{n+1} \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ . \\ y_{n+1} \end{bmatrix}$$

Program listing

01	LBL "POLFIT"	to use, simply XEQ "**POLFIT**"
02	"N=?"	prompts for the degree N of the polynomial
03	PROMPT ..	and waits for the user to enter N
04	1	add 1 to get the number of data points
05	+	N+1
06	1.001	the required multiplier
07	*	forms the matrix dimensions [N+1].00[N+1]
08	*"MX"*	specifies matrix MX to be created in X-Mem
09	**MATDIM**	creates and dimensions matrix MX in X-MEM
10	0	specifies first row, first column and ..
11	**MSIJ** ..	resets the row/column indexes
12	LBL 00	loop to ask for data & compute MX elements
13	**MRIJ**	recalls the current value of the indexes
14	*"X"*	forms the prompt to ask the user to enter xi
15	**AIP**	appends the index to the prompt
16	*"|-=?"*	appends "=?" to the prompt
17	PROMPT	prompts to enter xi and resume execution
18	ENTER^	fills the stack with the value of xi ..
19	ENTER^	in order to compute all powers of xi ..
20	ENTER^	from 1 to xi^n and store them in MX
21	1	initializes the value of xi^0 [i.e.: 1]
22	**MSR+**	stores it in MX and updates the indexes

23	LBL 01	loop to compute the powers of xi 24 * computes xi^j
25	**MSR+**	stores it in MX and updates the indexes
26	FC? 09	are we done with this row ?
27	GTO 01	not yet, go back for the next xi power
28	FC? 10	row done. Are we done with all rows?
29	GTO 00	not yet, go back to ask for the next xi
30	CLA	all rows done, MX complete. Make it current
31	**DIM?**	get its dimensions: [N+1].00[N+1]
32	INT	get N+1 (avoid using a register)
33	*"MY"*	specify vector MY to be created in X-MEM
34	**MATDIM**	creates and dimensions vector MY in X-MEM
35	LBL B	ask for yi data and store them in MY
36	0	specifies 1st element of the vector and ...
37	**MSIJ** ..	resets the index to the 1st element
38	LBL 02	loop for next data and store them in MY
39	**MRIJ**	recalls the current value of the index
40	"Y"	forms the prompt to ask for yi
41	**AIP** ..	appends the index to the prompt
42	*"/-=?"*	appends "=?" to the prompt
43	PROMPT	prompts the user to enter yi
44	**MSR+**	stores it in MY and updates the index
45	FC? 10	are we done with all elements?
46	GTO 02	not yet, go back to ask for the next yi
47	*"MX,MY"*	all yi stored. Specify MX,MY for the system
48	**MSYS**	solves the system for the coefficients
49	LBL C	retrieve and display each coeff.
50	0	specifies 1st element of the coeffs. vector
51	**MSIJ**	resets the index to the 1st coefficient
52	LBL 03	loop to retrieve the next coefficient
53	**MRIJ**	recalls the current value of the index
54	"A"	forms the prompt to display each coeff.
55	**AIP** ..	appends the index to the prompt
56	*"/-="*	appends "=" to the prompt
57	**MRR+**	retrieves the value of the current coeff.
58	ARCL X	appends the value to the prompt
59	PROMPT	shows the value to the user
60	FC? 10	are we done outputting all the coeffs?
61	GTO 03	not yet, go back for the next coefficient
62	END	all done. End of execution.

Notes

- As the Advantage ROM can work with matrices directly in X-Mem, **POLFIT** doesn't use any main RAM registers and so it will run even at SIZE 000. This has the added advantage (pun intended) of avoiding any register conflicts with other programs.
- POLFIT creates two matrices in X-Mem, namely [**MX**] and [**MY**], which aren't destroyed upon termination. Retaining [**MX**] allows the user to compute the coefficients of another polynomial using the same x data but different y data. In that case, the x data need not be entered again, only the new y data must be entered. Further, as the MX matrix is left in LU-decomposed form after the first fit, the second fit willproceed much faster. Retaining [**MY**] allows the user to employ the polynomial for interpolating purposes, root finding, numeric integration or differentiation, etc.
- Lines 2-11 prompt the user for the degree of the polynomial, then allocate the system matrix in Extended Memory (**MATDIM**) and reset the indexes (**MSIJ**).

- Lines 12-22 set up a loop that will fill up the rows of [**MX**]. Notice the use of the miscellaneous function AIP to build the prompt, and MSR+ to store the value and automatically advance the indexes to point to the next element.

- Lines 23-27 form a tight loop that computes each power of xi and uses **MSR+** to store it and advance the indexes. Flag 9 logs if we're done with the column in which case we would proceed to the next row. If so, Flag 10 is then checked to see if we're done with all the rows.

- Once the system matrix has been populated, lines 30-45 do likewise dimension, and populate the MY matrix, prompting the user for the required yi values. Then, once all the data have been input and both matrices are allocated and populated, lines 46-47 solve the system for the coefficients of the polynomial using **MSYS**.

- Finally, lines 48-59 establish a loop that labels and outputs all the coefficients.

Example

Rumor has it that the seemingly trigonometric function **y = cos(5 arccos x)** is actually a 5th-degree polynomial in disguise. Attempt to retrieve its true form.

If it is indeed a 5th-degree polynomial, we can retrieve its true form by fitting a 5th-degree polynomial to a set of 6 arbitrary data points (x,y). Any set with different x values (-1.0 <= x <= +1.0) will do, but for simplicity's sake we'll use x=0, 0.2, 0.4, 0.6, 0.8, and 1. Proceed like this:

- set Rad mode, 4 decimals:	XEQ "RAD", FIX 4	
- start the program:	XEQ "POLFIT"	"N=?"
- specify degree 5:	5 R/S	"X1=?"
- enter 1st x value:	0 R/S	"X2=?"
- enter 2nd x value:	0.2 R/S	"X3=?"
- enter 3rd x value:	0.4 R/S	"X4=?"
- enter 4th x value:	0.6 R/S	"X5=?"
- enter 5th x value:	0.8 R/S	"X6=?"
- enter 6th x value:	1 R/S	"Y1=?"
- enter 1st y value:	0, ACOS, 5, *, COS, R/S	"Y2=?"
- enter 2nd y value:	0.2, ACOS, 5, *, COS, R/S	"Y3=?"
- enter 3rd y value:	0.4, ACOS, 5, *, COS, R/S	"Y4=?"
- enter 4th y value:	0.6, ACOS, 5, *, COS, R/S	"Y5=?"
- enter 5th y value:	0.8, ACOS, 5, *, COS, R/S	"Y6=?"
- enter 6th y value:	1, ACOS, 5, *, COS, R/S	"a1=-1.0250E-9"
	R/S	"a2=5.0000"
	R/S	"a3=7.0867E-8"
	R/S	"a4=-20.0000"
	R/S	"a5=2.6188E-7"
	R/S	"a6=16.0000"

So, disregarding the very small coefficients due to rounding errors, the undisguised polynomial is:

P(x) = cos(5 arccos x) = 5 x – 20 x^3 + 16 x^5

You might want to execute now CAT"4 (or EMDIR), to see that the matrices used are still available so that you can redisplay the coefficients, solve for a new set of y values, or use the polynomial for interpolation, etc.

CAT"4	"MX	M036"	[the system matrix is 6x6 = 36 elements]
	"MY	M006"	[the coeff. matrix is 6x1 = 6 elementss]
	554.0000		[this value varies with your configuration]

4.4.5. Orthogonal Polynomial Fit. { OPFIT }

Orthogonal polynomials are a very advantageous method for polynomial regression. Not only it allows for a more progressive approach, but also the accuracy of the values so obtained is typically better. This program employs this method; even if it doesn't calculate any orthogonal polynomials explicitely.

Given m value pairs (xi, yi) and a maximum degree to explore (n), this program calculates the n(n+3)//2 polynomial coefficients of the corresponding n polynomials of degrees 1, 2, 3,… n that best fit the given data (therefore equivalent to the least squares method). It also obtains the determination coefficients and typical errors for each degree,

The method followed uses the construct **Y(x) = d0 P0(x) + d1 P1(x) + … dn Pn(x)** ; where p0, p1, … pn are the orthogonal polynomials corresponding to the entered data that satisfy the expression Σpi Pj = 0, for every i#j

The advantage of this approach is a better accuracy, as it avoids the resolution of the usual n linear systems, frequently ill-conditioned, that arise in the least squares method.

Example.- To check the program we took the following 11 value pairs from the polynomial

P(x) = x^4 – 2x^3 + 3x^2 –4 x +5

Xi	-3	-2	-1	0	1	2	3	4	5	6	7560
Yi	179	57	15	5	3	9	47	165	435	953	1839

Using the data above explore up to degree n=4, showing the correlation coefficients, the D-factors and the errors for each of the alternatives.

The results are all provided in the table below:

Degree (n)	Corrlt. (r^2)	Errors (e^2)	Determ. (d^2)	Coefficients
n = 1	R1=4,482218E-1	E0=3,295160E5 E1=1,818197E5	D0=3,370000E2 D1=1,228000E2	a0=9,140000E1 a1=1,228000E2
n =2	R2=9,000134E-1	E2=3,294720E4	D2=4,000000E1	a0=-1,486000E2 a1=-3,720000E1 a2=4,000000E1
n = 3	R3=9,821452E-1	E3=5,883429E3	D3=6,000000E0	a0=1,700000E1 a1=-7,200000E1 a2=4,000000E0 a3=6,000000E0
n = 4	R4=1,000000E0	E4=0,000000E0	D4=1,000000E0	a0=5,000000E0 a1=-4,000000E0 a2=3,000000E0 a3=-2,000000E0 a4=1,000000E0

Original author: **OPFIT** was written bu Eugenio Úbeda, and published in the UPLE. The version in the SandMatrix has only minimal changes made to it. It is by far the longest program in the module.

4.4.6. From Poles to Zeros... and back. { POLZER , PFE }

These two programs complete the applications section. The first one 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. The second program builds the partial fraction expansion for a polynomial given it its "standard" (or natural) form.

	Function	Description	Input / Output
1	**POLZER**	Zeros of transfer functions	Under program control
2	**PFE**	Partial Fraction Expansion	Under program control

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

P(x) = Σ [1 / (x-pi)] ; i= 1,2,... n

Which will be transformed into:

P(x) = Σ ai 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..... "
	"CFS? Y/N"
"**Y**"	a(4)=5,00000
R/S	a(3)=-40,00000
R/S	a(2)=105,00000
R/S	a(1)=-100,00000
R/S	a(0)=24,00000

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 **RTSN** , which will calculate 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	Z=0,35557
R/S	Z=1,45609
R/S	Z=2,54395
R/S	Z=3,64442

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

4.4.7. Partial Fraction Decomposition

In algebra, the partial fraction decomposition or partial fraction expansion of a rational fraction (that is a fraction such that the numerator and the denominator are both polynomials) is the operation that consists in expressing the fraction as a sum of a polynomial (possibly zero) and one or several fractions with a simpler denominator.

In symbols, one can use partial fraction expansion (where *ƒ* and g are polynomials) to change expression forms as shown below

$$\frac{f(x)}{g(x)} \quad >>> \quad \sum_j \frac{f_j(x)}{g_j(x)}$$

where gj (x) are polynomials that are factors of g(x), and are in general of lower degree. Thus, the partial fraction decomposition may be seen as the inverse procedure of the more elementary operation of addition of rational fractions, which produces a single rational fraction with a numerator and denominator usually of high degree. The full decomposition pushes the reduction as far as it will go: in other words, the factorization of g is used as much as possible. Thus, the outcome of a full partial fraction expansion expresses that fraction as a sum of fractions, where:

the denominator of each term is a power of an irreducible (not factorable) polynomial and the numerator is a polynomial of smaller degree than that irreducible polynomial. To decrease the degree of the numerator directly, the Euclidean division can be used, but in fact if *ƒ* already has lower degree than g this isn't helpful.

Implementation

POLZER may be an old program but **PFE** is a much more modern event, written by JM Baillard and published at: http://hp41programs.yolasite.com/part-frac-expan.php

Given a rational function $R(x) = P(x) / Q(x)$ with $Q(x) = [\ q_1(x)\ ]^{\mu 1}$ .... $[\ q_n(x)\ ]^{\mu n}$ and $\gcd(q_i , q_j) = 1$ for all i # j , this program returns the partial fraction expansion:

$$R(x) = E(x) + p_{1,1}(x) / [\ q_1(x)\ ]^{\mu 1} + p_{1,2}(x) / [\ q_1(x)\ ]^{\mu 1-1} + \ldots\ldots + p_{1,\mu 1}(x) / q_1(x)$$

$$+ \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots$$

$$+ p_{n,1}(x) / [\ q_n(x)\ ]^{\mu n} + p_{n,2}(x) / [\ q_n(x)\ ]^{\mu n-1} + \ldots\ldots\ldots + p_{n,\mu n}(x) / q_n(x)$$

where deg $p_{i,k}$ < deg q_i , and E(x) is the quotient in the Euclidean division P(x) = E(x) Q(x) + p(x) and p(x) is the remainder.

Data entry is a complicated affair but it has been automated – just follow the process carefully. It makes extensive use of the polynomial arithmetic routines **PPRD** and **PDIV**. Also the polynomial entry routine **PEDIT** is called several times...

The program prompts for the number of factors in the denominator, as well as for their degrees and multiplicities. It also prompts for the coefficients of the numerator polynomial and of each factor polynomial in the denominator; so you don't need to store those values manually prior to executing PFE.

Data output is not automated; therefore you'd need to interpret the control words returned in stack registers. Some guidelines will follow in the examples.

Example1. Calculate the partial fraction decomposition for R(x) below.

$$R(x) = P(x)/Q(x) = (6 x^5 - 19 x^4 +20 x^3 - 7 x^2 + 7 x + 10) / [(2 x^2 + x + 1).(x - 2)^2]$$

Keystrokes	Display	Result
XEQ "**PFE**"	"#DEN=?"	Input number of factors
2, R/S	"NUM#=?"	inputs degree of numerator
5, R/S	"Σ(aK*X^K)"	Reminder of convention
	"a5= ?"	coefficients data entry
6, R/S	"a4= ?"	
19, CHS, R/S	"a3= ?"	
20, R/S	"a2= ?	
7, CHS, R/S	"a1=?"	
7, R/S	"a0=?"	
10, R/S	"Q1#=?"	Input degree of Q1 in den.
2, R/S	"Σ(aK*X^K)"	Reminder of convention
	"a2=?"	
2, R/S	"a1=?"	
1, R/S	"a0=?"	
1, R/S	"Q2#=?"	
1, R/S	"Σ(aK*X^K)"	Reminder of convention
	"a1=?"	
1, R/S	"a0=?	
2, CHS, R/S	"XP^μ"	time to enter the multiplicities now
	"a1= ?"	exponent of first factor
1, R/S	"a0= ?"	exponent of second factor
2, R/S	flying goose…	beep sounds
	"E(x)"	informs that E(x) follws
	"Σ(aK*X^K)"	Reminder of convention
	"a1=3"	
R/S	"a0=1"	end of data output.

There are three control words placed registers R05, R06, and R15 upon completion, as follows:

1. The cnt'l word stored in R15 is for the Quotient polynomial, E(x)
2. The cnt'l word in R05 gives the entire register range for the coefficients of all the pi(x) polynomials – the numerators of the expanded fractions. It needs to be interpreted depending on the denominators qi(x) are polynomials of degree 1 or polynomials pf degree 2 with negative discriminant. The contents of these registers are to be read

- by groups of 1 number if deg qj = 1 the numerators are constants
- by groups of 2 numbers if deg qj = 2 the numerators are polynomials of degree 1
- by groups of 3 numbers if deg qj = 3 the numerators are polynomials of degree 2 , and so on ....

3. The third in R06 is for an alternative solution using a new reminder p(x)

Thus in this case registers R16 and R17 contain the coefficients for **E(x)** = 3x + 1;
And registers R33 – R36 for the denominator polynomials: (which must be three of them!)

p1,1(x) = 2x + 3 ; p2,1(x) = 4 ; p2,2(x) = 5

Thus the final result is as follows:

R(x) = **E(x)** + **p1,1(x)** /(2x^2 + x +1) + **p2,1(x)** / (x-2)^2 + **p2,2(x)** / (x-2)

Or alternatively using the data in registers R18 – R21 (cnt'l word in Z):

p(x) = 12 x^3 – 12x^2 – 5x +6 ; and thefore:

R(x) = **E(x)** + **p(x)** /Q(x)

Example 2.- Calculate the partial fraction decomposition for R(x) below.

R(x) = P(x)/Q(x) = x^5 /(3 x^2 + 1)2

The three control words returned are:

Z: 18.021 with: R18=-2/3, R19= 0, R10 =-1/9, R21 =0
Y: 28.031 with R29=1/9, R29=0, | R30=-2/9, and R31=0
X: 16,017 with: R16 = 1/9 and R17 = 0

The range in Y must be split as p1,2 = x/9 x + 0; and p2,2 = -2x/9 + 0

Therefore:
R(x) = E(x) + p1,2(x)/(3x^2 + 1)^2 + p2,2(x)/(3x^2 + 1)

All in all a powerful program, which flexibility requires some careful attention to the details involved.

Note:- you can check another Partial Fraction expansion program (by Narmwon Kim) available at the HP-41 archive site, which features a simpler user interaction and data entry/output, but at the expense of more limited functionality. It is also less general-purpos, and more geared towards control system applications.

http://www.hp41.org/LibView.cfm?Command=View&ItemID=776

Appendix –M. MCODE listings for LU? And ^MROW .

There are a few new M-Code functions in the SandMatrix that make direct usage of the module's subroutines. A representative example is given below, showing the very short routine LU? – that checks whether the matris is in its decomposed form – simply by reading the appropriate digit in the matrix header register.

1	LU?	Header	A5FA	0BF	"?"	
2	LU?	Header	A5FB	015	"U"	
3	LU?	Header	A5FC	00C	"L"	
4	LU?	**LU?**	**A5FD**	379	PORT DEP:	Jumps to Bank_2
5	LU?		A5FE	03C	XQ	adds "4" to [XS]
6	LU?		A5FF	1D9	->A5D9	[LNCH0]
7	LU?		A600	388	<parameter>	B788
8	LU?		A601	00B	JNC +01	
9	LU?		A602	100	ENROM1	restore bank-1
10	LU?		A603	0B0	C=N ALL	header register
11	LU?		A604	25C	PT= 9	LU digit
12	LU?		A605	2E2	?C#0 @PT	
13	LU?		A606	0B9	?NC GO	False
14	LU?		A607	05A	->162E	[SKP]
15	LU?		A608	065	?NC GO	True
16	LU?		A609	05A	->1619	[NOSKP]

Lastly, and just in case you though that functions **PMTM** and **PMTP** are actually not a big deal (which would be the logical conclusion if you only look at their FOCAL program listing) – here is in all its gory detail the listing for its MCODE-heart, function **^MROW**.

I'll spare you the more onerous details, but suffice it to say that it was an involved assignment. And don't forget that another function is also used to support the matrix prompt mode: ANUMDL – although in this case I just had to copy HP's code from the HP-IL Development Module (thanks HP! :-)

1	^MROW	Header	B658	097	"W"	
2	^MROW	Header	B659	00F	"O"	
3	^MROW	Header	B65A	012	"R"	**Input Matrix Row**
4	^MROW	Header	B65B	00D	"M"	
5	^MROW	Header	B65C	01E	"^"	Ángel Martin
6	^MROW	**^MROW**	**B65D**	**0C4**	**CLRF 10**	start anew: no CHS yet
7	^MROW		B65E	184	CLRF 11	start anew: no commas yet
8	^MROW		B65F	344	CLRF 12	start anew: no digits yet
9	^MROW		B660	0F8	READ 3(X)	
10	^MROW		B661	070	N=C ALL	
11	^MROW		B662	345	?NC XQ	Clears Alpha
12	^MROW		B663	040	->10D1	[CLA]
13	^MROW		B664	215	?NC XQ	Build Msg - all cases
14	^MROW		B665	0FC	->3F85	[APRMSG2]
15	^MROW		B666	212	"R"	
16	^MROW		B667	0B0	C=N ALL	row number in BCD format
17	^MROW		B668	37C	RCR 12	move the MSB to C(0)
18	^MROW		B669	21C	PT= 2	
19	^MROW		B66A	010	LD@PT- 0	
20	^MROW		B66B	2D0	LD@PT- B	add colon to digit
21	^MROW		B66C	3E8	WRIT 15(e)	write it in display (9-bit)
22	^MROW		B66D	355	?NC XQ	blank space to LCD
23	^MROW		B66E	03C	->0FD5	DSPL20
24	^MROW		B66F	33D	?NC GO	Input List in Alpha
25	^MROW		B670	112	->44CF	[ALIST]

Not such a big deal, you keep saying? Well, let's have a look at the remaining part in the Libary#4

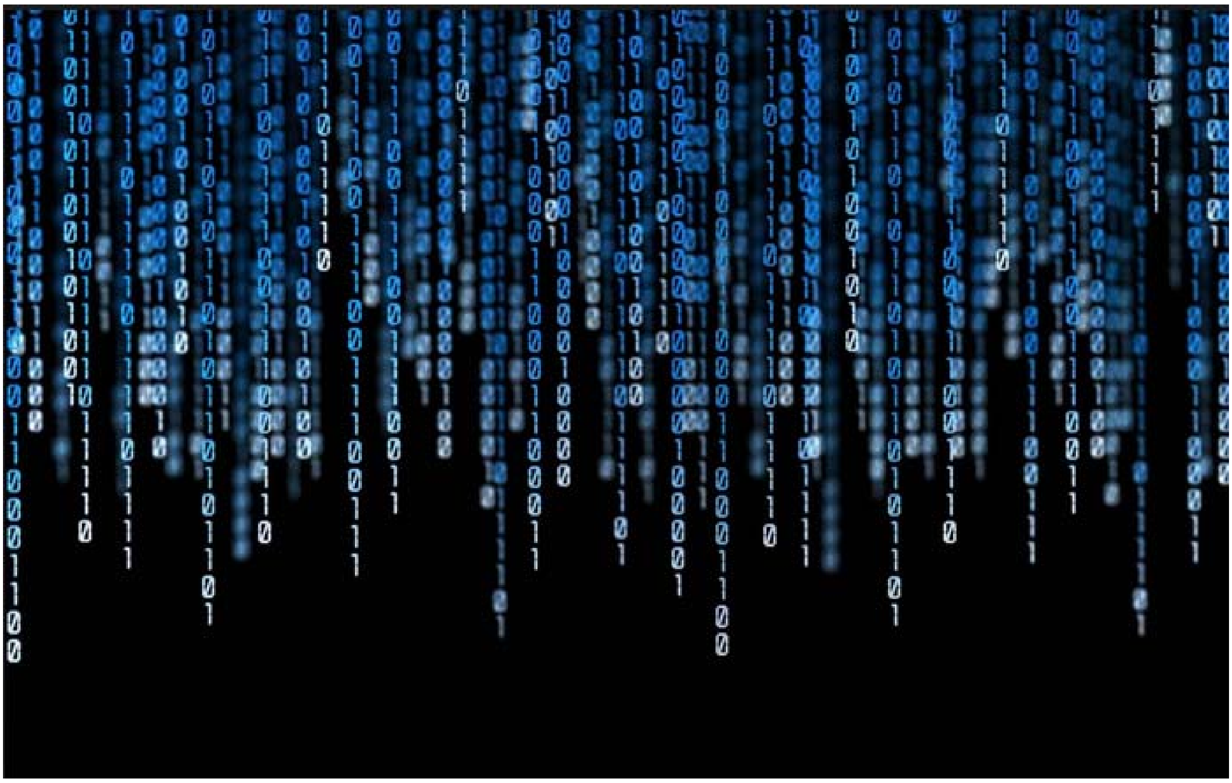
1	ALIST	BCKARW	44CD	055	?NC GO	Delete char plus logic
2	ALIST		44CE	116	->4515	[DELCHR]
3	ALIST	ALIST	44CF	115	?NC XQ	Partial Data Entry!
4	ALIST		44D0	038	->0E45	[NEXT1]
5	ALIST		44D1	3E3	JNC -04	[BCKARW]
6	ALIST		44D2	00C	?FSET 3	numeric input?
7	ALIST		44D3	093	JNC +18d	NO, KEEP LOOKING
8	ALIST		44D4	0BE	A<>C MS	recall LS digit from A[13]
9	ALIST		44D5	130	LDI S&X	
10	ALIST		44D6	003	CON:	pre-load the numeric mask
11	ALIST		44D7	2FC	RCR 13	move it to C[S&X]
12	ALIST		44D8	3E8	WRIT 15(e)	write it in display (9-bit)
13	ALIST		44D9	348	SETF 12	enable SPACE
14	ALIST	TOALPH	44DA	39C	PT= 0	
15	ALIST		44DB	058	G=C @PT,+	
16	ALIST		44DC	149	?NC XQ	Disable PER, enable RAM
17	ALIST		44DD	024	->0952	[ENCP00]
18	ALIST		44DE	051	?NC XQ	
19	ALIST		44DF	0B4	->2D14	[APNDNW]
20	ALIST	GOBACK	44E0	042	C=0 @PT	
21	ALIST		44E1	058	G=C @PT,+	reset PTEMP bits
22	ALIST		44E2	3D9	?NC XQ	Enable Display (not cleared)
23	ALIST		44E3	01C	->07F6	[ENLCD]
24	ALIST	ANCHOR1	44E4	35B	JNC -21d	ONE PROMPT
25	ALIST		44E5	28C	?FSET 7	decimal key pressed?
26	ALIST		44E6	03B	JNC +07	NO, KEEP LOOKING
27	ALIST		44E7	18C	?FSET 11	been used already?
28	ALIST		44E8	3E7	JC -04	ONE PROMPT
29	ALIST		44E9	188	SETF 11	no more radix (unless deletion)
30	ALIST		44EA	10D	?NC XQ	adds proper radix sign
31	ALIST		44EB	114	->4543	[RADIX4]
32	ALIST	ANCHOR2	44EC	373	JNC -18d	[TOALPH]
33	ALIST		44ED	0B0	C=N ALL	PRESSED KEY CODE
34	ALIST		44EE	106	A=C S&X	
35	ALIST		44EF	130	LDI S&X	
36	ALIST		44F0	030	CON:	ENTER^ keycode [030]
37	ALIST		44F1	366	?A#C S&X	
38	ALIST		44F2	04F	JC +09	
39	ALIST		44F3	34C	?FSET 12	digits input already?
40	ALIST	ANCHOR1	44F4	383	JNC -16d	ONE PROMPT
41	ALIST		44F5	0C4	CLRF 10	clear CHS flag
42	ALIST		44F6	184	CLRF 11	allow RADIX
43	ALIST		44F7	344	CLRF 12	set SPACE flag
44	ALIST		44F8	355	?NC XQ	add space to LCD
45	ALIST		44F9	03C	->0FD5	[DSPL20]
46	ALIST		44FA	393	JNC -14d	add to Alpha
47	ALIST		44FB	130	LDI S&X	
48	ALIST		44FC	370	CON:	R/S keycode [370]
49	ALIST		44FD	366	?A#C S&X	terminate digit entry
50	ALIST		44FE	07B	JNC +15d	[WAYOUT]
51	ALIST		44FF	130	LDI S&X	
52	ALIST		4500	230	CON:	CHS keycode [230]
53	ALIST		4501	366	?A#C S&X	
54	ALIST		4502	023	JNC +04	
55	ALIST		4503	265	?NC XQ	Blink Display - pass #2
56	ALIST		4504	020	->0899	[BLINK1]
57	ALIST		4505	37B	JNC -17d	ONE PROMPT
58	ALIST		4506	0CC	?FSET 10	been used already?
59	ALIST		4507	3F7	JC -02	ONE PROMPT
60	ALIST		4508	0C8	SETF 10	first time
61	ALIST		4509	130	LDI S&X	
62	ALIST		450A	02D	"-"	appends "-"
63	ALIST		450B	3E8	WRIT 15(e)	9-bit LCD write
64	ALIST		450C	303	JNC -32d	[TOALPH]
65	ALIST	WAYOUT	450D	3DD	?NC XQ	Left-justify LCD
66	ALIST		450E	0AC	->2BF7	[LEFTJ]

ALIST		450F	161	?NC XQ	Clear LCD and reset things
ALIST		4510	124	->4958	[EXIT3]
ALIST		4511	175	?NC XQ	Adjust F10 Status
ALIST		4512	114	->455D	[ADJF10]
ALIST		4513	31D	?NC GO	Normal Function ReturnKB
ALIST		4514	002	->00C7	[NFRKB]
ALIST	**DELCHR**	**4515**	**3B8**	**READ 14(d)**	to delete rightmost chr
ALIST		4516	158	M=C ALL	save it for later
ALIST		4517	149	?NC XQ	Disable PER, enable RAM
ALIST		4518	024	->0952	[ENCP00]
ALIST		4519	178	READ 5(M)	
ALIST		451A	2EE	?C#0 ALL	anything in Alpha?
ALIST		451B	**037**	**JC +06**	yes, go on
ALIST		451C	104	CLRF 8	no, abort if empty
ALIST		451D	1B1	?NC XQ	Mainframe Message
ALIST		451E	070	->1C6C	[MSGA]
ALIST		451F	03C	"NULL"	from table
ALIST	fixed bug	**4520**	**37B**	**JNC -17d**	Reset everything and leave
ALIST		4521	2E5	?NC XQ	remove last Alpha char
ALIST		4522	110	->44B9	[ABSP4]
ALIST		4523	198	C=M ALL	recall deleted char value
ALIST		4524	106	A=C S&X	store in A for comparisons
ALIST		4525	130	LDI S&X	check for SPACE
ALIST		4526	020	**"space"**	<space>
ALIST		4527	0AD	?NC XQ	complete the logic
ALIST		4528	114	->452B	[CHUNK4]
ALIST		4529	381	?NC GO	repeat the prompt
ALIST		452A	112	->44E0	[GOBACK]
ALIST	**CHUNK4**	**452B**	366	?A#C S&X	carry if different
ALIST		452C	01F	JC + 03	
ALIST		452D	348	SETF 12	allow new space entry
ALIST		452E	**0A3**	**JNC +20d**	BAIL OUT
ALIST		452F	130	LDI S&X	check for "-" chr
ALIST		4530	02D	**"-"**	"-" char value
		4531	366	?A#C S&X	carry if not "-"
Executed within [DELCHR]		4532	02F	JC + 05	
an opportunistic routine		4533	**34C**	**?FSET 12**	is there SPACE chr?
just grouping common code		4534	**017**	**JC +02**	
		4535	0C4	CLRF 10	allow new "-" entry
ALIST		4536	**063**	**JNC +12d**	BAIL OUT
ALIST		4537	198	C=M ALL	recall deleted char value
ALIST		4538	3D8	C<>ST XP	Got a radix? If so, we neet to
ALIST		4539	14C	?FSET 6	replace it without comma
ALIST		453A	**043**	**JNC +08**	
ALIST		453B	3D9	?NC XQ	Enable Display (not cleared)
ALIST		453C	01C	->07F6	[ENLCD]
ALIST		453D	144	CLRF 6	remove the radix value
ALIST		453E	284	CLRF 7	(both if need be)
ALIST		453F	3D8	C<>ST XP	recall deleted char value
ALIST		4540	**3E8**	**WRIT 15(e)**	write it in display
ALIST		4541	**184**	**CLRF 11**	Re-allow comma writing
ALIST		4542	**3E0**	**RTN**	
ALIST	**RADIX4**	**4543**	149	?NC XQ	Disable PER, enable RAM
ALIST		4544	024	->0952	[ENCP00]
ALIST		4545	3B8	READ 14(d)	put F28 to F9
		4546	2BC	RCR 7	
transfer staus of UF28 to F9,		4547	**248**	**SETF 9**	
adds the converted crh code		4548	1EE	C=C+C ALL	comma or period ?
to the LCD and prepares ALPHA		4549	013	JNC +02	overflows if COMMA (cf28)
		454A	**244**	**CLRF 9**	comma = CF 28
ALIST		454B	3D9	?NC XQ	Enable Display (not cleared)
ALIST		454C	01C	->07F6	[ENLCD]
ALIST		454D	**3B8**	**READ 14(d)**	read right
ALIST		454E	3D8	C<>ST XP	
ALIST		454F	148	SETF 6	
ALIST		4550	**24C**	**?FSET 9**	comma or period ?
ALIST		4551	013	JNC +02	
ALIST		4552	288	SETF 7	should replace the last chr
ALIST		4553	3D8	C<>ST XP	with the same one w/ radix
ALIST		4554	**3E8**	**WRIT 15(e)**	9-bit LCD write
ALIST		4555	130	LDI S&X	
ALIST		4556	02C	**","**	appends "," [02C]
ALIST		4557	**24C**	**?FSET 9**	
ALIST		4558	**360**	**?C RTN**	no need, return
ALIST		4559	226	C=C+1 S&X	
ALIST		455A	226	C=C+1 S&X	appends "." [02E]
ALIST		455B	**3E0**	**RTN**	

The End.

This concludes the SandMatrix Manual – Hope you have found it useful and interesting enough to keep as a reference. Better yet, go ahead and write a few more functions on your own. A few suggestions are:

- Program to calculate Eigenvectors from Eigenvalues
- General-purpose p-th. root of a matrix
- General-purpose Logarithm of a matrix
- Anything else you feel like going for!



Note: Make sure that revision "H" (or higher) of the Library#4 module is installed.