SANDMATRIX_5Y+

Matrix Extensions for the HP-41.

User’s Manual and Quick Reference Guide

Written and programmed by Ángel M. Martin

January 18th, 2021

This compilation revision 5.1.02

Acknowledgments.-

Documentation wise, this manual begs, steals and borrows from many other sources – in particular from the HP-41 Advantage Manual. Not so much 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; Ulrich Deiters and Thomas Klemm. 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_5Y.

1. Introduction: the SandMatrix 4

2. Lower Page Functions in Detail

2.1. SandMatrix 4 Group

2.2. Matrix-101

2.3. Matrix Functions

3. Upper Page Functions in Detail

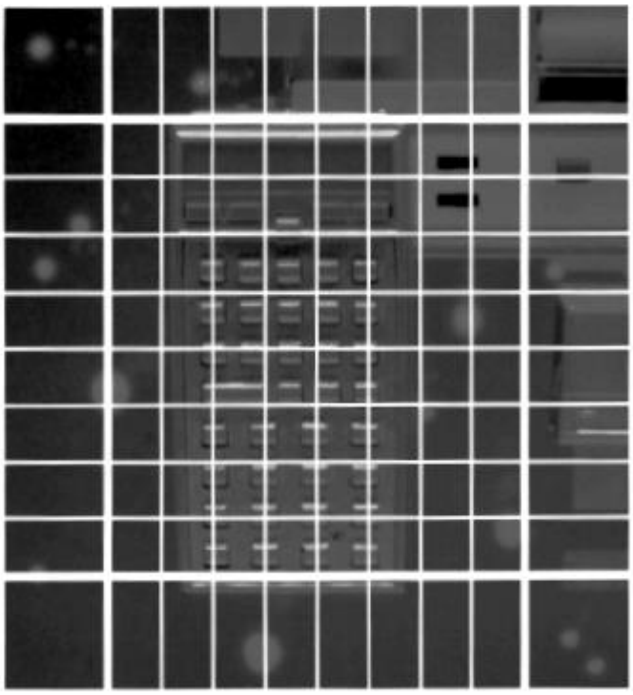
3.1. Advanced Matrix

3.2. Polynomials and Linear Algebra.

3.3. Managing Polynomials

3.4. Applications of Polynomials

4. Geometry and Vector Analysis



Note: Make sure that revision "R58" (or higher) of the Library#4 module is installed.

SandMatrix_4 Module – Revision “5Y+”
Matrix Extensions for the HP-41 System.

0. Preamble to latest revisions.

The latest revision of the SandMatrix rounds up the module with subtle architectural enhancements, as well as a few additional new functions to complete the pack. Notable are the all-MCODE versions of polynomial evaluation and its derivatives, the new Orthogonal Polynomial functions, and the addition of the FOCAL examples from the manual now available to the user in the function set. Simplification of the code allowed for optimal allocation of the routines, freeing up space for the new additions.

Revision “M” of the SandMatrix included significant additions to its predecessor – both in contents as well as in what relates to its internal architecture. Short on the heels of revisions “K” and “L”, the author figured out a way to enhance the original bank-switching implementation (based on HP-‘s Advantage Module), adding support for the Hepax-based model to it; and thereby enabling the utilization of the second bank in the lower page for other purposes. This freed up a substantial amount of room that now hosts the **3D Vector Calculator** ROM almost in its entirety – using a secondary FAT and auxiliary function launchers like those present in the SandMath module.

The benefits are obvious: *all 3D Vector functions are now available within the SandMatrix*, saving a page in the calculator configuration ports. It also facilitated further consolidation of functions, removing the poor-man FOCAL implementation of the same capabilities from the SandMatrix. The net result is a much more capable module that now *includes all the main components of Algebra*: Vectors, Polynomials and Matrices.

The 3D-Vector functions can be executed by their names (using function “**ΣV$**”), or using their indexes within the secondary FAT (using function “**ΣV#**”). These two functions are further connected to the main module function launcher ΣML using the ALPHA and USER keys respectively at its prompt. Note also that the “ΣML” prompt toggles cyclically amongst three personalities (or modes) upon repeated pressing of the Σ+ key as follows: “M:_” for matrix functions, “P:_” for polynomial functions, and ‘V:_” for vector functions. These refer to the dedicated keyboards as can be seen in the custom overlays shown later in the manual.

Because prompting functions cannot be easily located in secondary FATs, a few of the vector functions have also been placed in the main FAT of the module. These are **VRCL**, **VSTO**, **V<>**, and **VVIEW** – which use the prompt to select the register set needed in their operation. To make room for their entries, the non-prompting auxiliary functions (mainly dealing with ALPHA string management) have been relegated to the auxiliary FAT. The FOCAL programs that use them have also been modified accordingly to reflect the new arrangements.

Like it was the case in the Vectors ROM, the coordinate transformation and vector calculator programs **TR** and **VC** from the Advantage Pac have been re-written using the new 3D-Vectors functions included in the module – and included in the main FAT. User instructions and capabilities are identical, but its execution speed is largely faster, as they take advantage of the MCODE implementation of the underlying functions. Besides, a few other new geometry functions also went to the auxiliary FAT - completing the coordinate transformations and 2D/3D distances function set. They’ll be described in section 4 of the manual.

Note that all these important additions required making some changes to the Library#4 – make sure you use revision “R58” or higher in conjunction with the SandMatrix revision “Y”.

Note: Make sure that revision “R58” (or higher) of the Library#4 module is installed.

1. Introduction.

The release of the CCD Module by W&W in 1983 provided for the first time convenient and reliable tools for matrix algebra in the 41's platform. 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 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 previous revisions of the SandMatrix were very conventional. Back then there were no auxiliary FATs with sub-functions a la SandMath, and not even a dedicated function launcher with an *alternate keyboard*. All that is now completely changed and you can find all those ammenieties and a few more surprises in revisions "M,N, and Y". Many 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. 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.

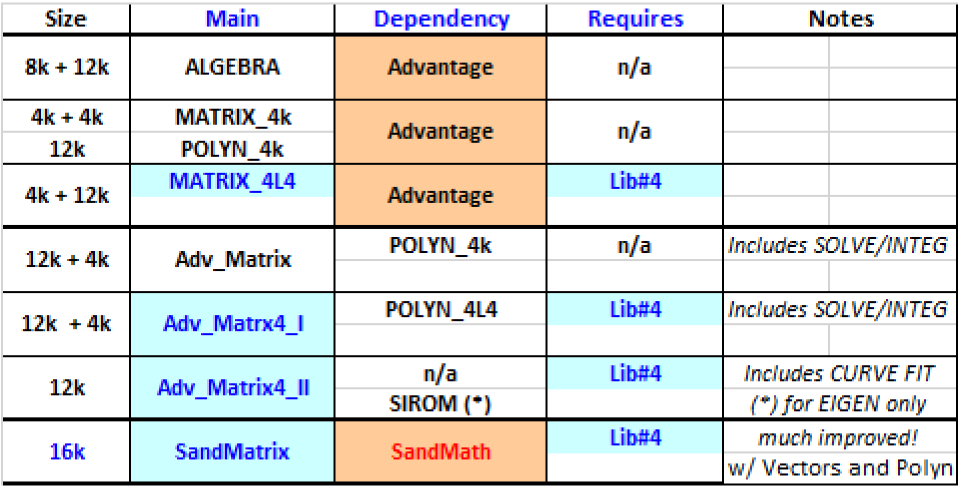
The solution to this riddle came only with the latest revision of the SandMath 3x3, which added a third bank with Solve and Integrate – plus an important consolidation of functions in its auxiliary FAT. This really cleared things out for the SandMatrix, in that the only dependencies left are the Library#4 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 kind of replaces all previous versions of the "Advanced Matrix ROM", the "Matrix ROM", and the "Polynomial ROM" (not counting the one co-produced with 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 + 12k	ALGEBRA	Advantage	n/a	
4k + 4k 12k	MATRIX_4k POLYN_4k	Advantage	n/a	
4k + 12k	MATRIX_4L4	Advantage	Lib#4	
12k + 4k	Adv_Matrix	POLYN_4k	n/a	*Includes SOLVE/INTEG*
12k + 4k	Adv_Matrx4_I	POLYN_4L4	Lib#4	*Includes SOLVE/INTEG*
12k	Adv_Matrix4_II	n/a SIROM (*)	Lib#4	*Includes CURVE FIT* *(*) for EIGEN only*
16k	SandMatrix	SandMath	Lib#4	*much improved!* w/ Vectors and Polyn

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	SandMath 4x4	Digit Pac	Includes 16C Emulator
Solve & Integrate	SandMath 4x4	Solve & Integrate ROM	Fully embedded
Curve Fitting	SandMath 4x4	AECROM	Fully embedded
Complex Operations	HP-41Z	-	Dedicated 8k ROM
Vectors / Coordinates	*Now also included !*	Vector Calculator ROM	Fully Embedded
Differential Equations	Diffeq ROM	Math Pac	Dedicated 8k ROM
Time Value of Money	SandMath 4x4	TVM Module	Fully embedded

Note: Make sure that revision “R58” (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	ΣML _	Matrix Function Launcher	Prompts for function	Executes Function	*Ángel Martin*
3	ΣDST _	Distance Functions Launcher	Prompts "PP PL LL PPL"	Executes function	*Ángel Martin*
4	ΣV# _ _ _	Sub-Function Launcher	Prompts for index	Executes Function	*Ángel Martin*
5	ΣV$ _	Sub-Function Launcher	Prompts for name	Executes Function	*Ángel Martin*
6	"MATRX"	‘Easy” Matrix Program	Driver for Major Matrix Ops.	Under prgm control	*HP Co.*
7	"TR"	Coordinate Transformation	Prompts for Data	Under prgm control	*HP Co.*
8	"VC"	Vector Calculator	Prompts "DP CP <) M UV"	Under prgm control	*HP Co.*
9	V<> _ _	Vector Swap	Vector register in Memory	Swaps vector with with V1	*Ángel Martin*
10	VRCL _ _	Vector Recall	Vector register in Memory	Recalls vector to V1	*Ángel Martin*
11	VSTO _ _	Vector Storage	Vector register in Memory	Stores V1 in Memory	*Ángel Martin*
12	VVIEW _ _	Shows Vector Components	Vector register in Memory	Shows vector components	*Ángel Martin*
13	V*VN	N-dimensional Dot product	Prompts for coeffs	Result in Matrix	*Ángel Martin*
14	-CCD MTRX	*Checks for LU, Square Matrix*	*MNAME in Alpha*	*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	Column 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*
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	**“CMDET”**	Complex Matrix Determinant	MNAME in ALPHA	Up to degree 4	*Ángel Martin*
1	**-ADV MATRIX**	*Section Header*	*none*	*Displays "Not Square"*	*Ángel Martin*
2	**CMR**	Recall complex element	Complex i,j in X	Puts value in {Y,X}	*Ángel Martin*
3	**CMS**	Store Complex element	Complex i,j in X, value in {Z,Y}	Leaves pointer at (2i, 2j)	*Ángel Martin*
4	**CMTRC**	Complex Matrix Trace	Mname in ALPHA	Result in {X,Y}	*Ángel Martin*
5	**I<>J**	Swaps indexes	iii,jjj in X	j,00i in X, i00j in LastX	*Ángel Martin*
6	**IMC**	Input Matrix by Columns	"OP1" in Alpha	Inputs elements by columns	*Ángel Martin*
7	**IMR**	Input Matrix by Rows	"OP1" in Alpha	Inputs elements by rows	*Ángel Martin*
8	**LU?**	Tests for LU-decomposed	MName in Alpha	YES/NO, Do if True.	*Ángel Martin*
9	**M^1/N**	p-th. root of a Matrix	"OP1" in Alpha, p in X	Matrix replaced by its root	*Ángel Martin*
10	**M^2**	Matrix Square	"OP1" in Alpha	Matrix replaced by [M][M]	*Ángel Martin*
11	**MAT=**	Copy Matrix	"OP1,RES" in Alpha	Copies matrix A into B	*Ángel Martin*
12	**MATP**	Driver for M*M	Driver for M*M	Under prgm control	*Ángel Martin*
13	**MCON**	Constant	"OP1" in Alpha, x in X	Makes all elements =x	*Ángel Martin*
14	**MDPS**	Diagonal Product Sum	"OP1" in Alpha	Sum of diagonal products	*Ángel Martin*
15	**MEXP**	Matrix Exponential	"OP1" in Alpha	Matrix replaced by exp(M)	*Ángel Martin*
16	**MFIND**	Element finder	"OP1" in Alpha, x in X	Element pointer if found	*Ángel Martin*
17	**MIDN**	Identity Matrix	"OP1" in Alpha	Makes it Identity Matrix	*Ángel Martin*
18	**MLIE**	Matrix Lie Product	"OP1,OP2,RES" in Alpha	[A][B] - [B][A]	*Ángel Martin*
19	**MLN**	Matrix Natural Log	"OP1" in Alpha	Matrix replaced by LN(M)	*Ángel Martin*
20	**MPINV**	Matrix Pseudo-Inverse	MName in Alpha	Pseudo-Inverse name in Alpha	*Ángel Martin*
21	**MPOL**	Matrix polynomial	Mname in Alpha, Cnt'l word in X	Calculates P([A])	*Ángel Martin*
22	**MPWR**	Matrix Integer Power	"OP1" in Alpha, N in X	Matrix replaced by [M]^INT(x)	*Ángel Martin*
23	**MRDIM**	Matrix Redimension	"OP1" in Alpha, dim in X	Matrix redimensioned	*Ángel Martin*
24	**MSORT**	Sorts matrix elements	"OP1" in Alpha	Matrix Elements sorted	*Ángel Martin*
25	**MSQRT**	Matrix Square Root	"OP1" in Alpha	Matrix replaced by SQRT([M])	*Ángel Martin*
26	**MSZE?**	Matriz Size	"OP1" in Alpha	Matrix size in X	*Ángel Martin*
27	MTRACE	Matrix Trace	"OP1" in Alpha	Trace in x	*Ángel Martin*
28	**MXIJ**	Exchanges element pointer	Pointer in matrix header	Returns new pointer to X	*Ángel Martin*
29	**MZERO**	Zeroes a Matrix	"OP1" in Alpha	All elements zeroed	*R.D. Kendon*
30	**OMC**	Output Matrix by Columns	"OP1" in Alpha	Shows elements by columns	*Ángel Martin*
31	**OMR**	Output Matrix by Rows	"OP1" in Alpha	Shows elements by rows	*Ángel Martin*
32	**PMTM**	Prompts for Matrix	"OP1" in Alpha	Prompts for complete Rows	*Ángel Martin*
33	**SQR?**	Tests for Square Matrix	MName in Alpha	YES/NO, Do if True.	*Ángel Martin*
34	**ΣIJJI**	Sum of crossed products	"OP1" in Alpha	Σ[aij*aji] in X	*Ángel Martin*
35	**-ADV POLYN**	*Section Header*	*none*	*Displays "Σ(ak*X^k)"*	*Ángel Martin*
36	**ΣPIPJ**	Symmetric Polyn	Cnt’l word in X	Sums coeffs. Products	*JM Baillard*
37	**BAIRS**	Bairstow Method	Cntl word in Z, guesses in Y,X	shows results	*JM Baillard*
38	**CHRPOL**	Characteristic Polynomial	"OP1" in Alpha	Characteristic Pol Coefs/Roots	*Ángel Martin*
39	**EIGEN**	Eigen Values by SOLVE	Under prgm control	Eigen Values by Solve	*Ángel Martin*
40	**EV3X3**	Eigen values 3x3	Prompts Matrix Elements	Eigen Values by Formula	*Ángel Martin*
41	**"LL3"**	3D Line-to Line distance	Under prgm control	Result in X	*Ángel Martin*
42	**"PL3"**	3D Point-to-Line distance	Under prgm control	Result in X	*Ángel Martin*
43	**"PPL3"**	3D Point-to-Plane distance	Under prgm control	Result in X	*Ángel Martin*

44	**MPT**	Matrix Index prompt	Adds i,j and “=” to ALPHA	Prompts and waits	*HP Co.*
45	**P+P**	Polynomial Addition	Driver for PSUM w/CF 01	shows results	*Ángel Martin*
46	**P-P**	Polynomial Substraction	Driver for PSUM w/SF 01	shows results	*Ángel Martin*
47	**P*P**	Polynomial Multiplication	Driver for PPRD	shows results	*Ángel Martin*
48	**P/P**	Polynomial Division	Driver for PDIV	shows results	*Ángel Martin*
49	**PCPY**	Copy of Polynomial	from, to cntl words in Y,X	polynomial copied	*JM Baillard*
50	**PDIV**	Euclidean Division	cntl words in Y and X	cntl words remainder & result	*JM Baillard*
51	**PEDIT**	Polynomial Editor	cntl word in X	prompts for each coeff value	*Ángel Martin*
52	**PFE**	Partial Fraction Expansion	Under prgm control	see description to decode	*JM Baillard*
53	**PMTP**	Prompts for Polynomial	cntl word in X	prompts for complete list	*Ángel Martin*
54	**POLFIT**	Polynomial Fit	Under prgm control	calculates polynomial fit	*Valentín Albillo*
55	**POLINT**	Aitken Interpolation	Under prgm control	interpolation made	*Ulrich Deiters*
56	**“POLZR”**	From Poles to Coeffs	Under prgm control	shows Polynomial coeffs	*Martin-Baillard*
57	**PPRD**	Polynomial Product	cntl words in Z, Y, bbb in X	cntl word result in X	*JM Baillard*
58	**PRMF**	Prime Factors Decomposition	number in X	prime factors in XM Matrix	*Ángel Martin*
59	**PROOT**	Polynomial Roots	Under prgm control	Shows all roots	*Ángel Martin*
60	**PSUM**	Polynomial Sum	cntl words in Z, Y; bbb in X	cntl word result in X	*JM Baillard*
61	**PVAL**	Polynomial Evaluation	Cntl word in Y, x in X	Result in X	*JM Baillard*
62	**PVIEW**	Polinomial View	Cntl word in X	Sequential listing of coeffs	*Ángel Martin*
63	**QUART**	Quartic Equation Roots	coeffs in Stack (a4=1)	shows results	*Martin-Baillard*
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.

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.

Sub-functions in the secondary FAT.

The Vector Calculator and remaining subfunctions come next. Note the three sections in the listing, with the second and third groups of auxiliary functions covering polynomials and other geometry applications. Note also the new **Matrix Catalogs** at the very end of the table.

#	Function	Description	Input	Output	Author
0	**-V CALC**	*Section Header*	*none*	*Triggers CAT+*	*Ángel Martin*
1	**A+V**	Vector addition	Data in V1 and V2	Sum vector in V1	*Ángel Martin*
2	**A-V**	Vector Subtraction	Data in V1 and V2	Difference vector in V1	*Ángel Martin*
3	**A*V**	Vector Dot Product	Data in V1 and V2	Result in X-reg	*Ángel Martin*
4	**AXV**	Vector Cross Product	Data in V1 and V2	Product vector in V1	*Ángel Martin*
5	**A/V**	Vector Division	Data in V1 and V2	Division vector in V1	*Ángel Martin*
6	**EXSCR**	Alpha Exchange	Vector in V2	Swapped with scratch	*Ángel Martin*
7	**LASTV**	Last Vector recall	none	V1 pushed to V2, LastV to V1	*Ángel Martin*
8	**X*V**	Multiplication by Scalar	Data in stack	Result in V1	*Ángel Martin*
9	**V<>A**	Swaps vector levels	Data in V1 and V2	Swaps vectors	*Ángel Martin*
10	**VADST**	Distance between vectors	Data in V1 and V2	Distance in X-reg	*Ángel Martin*
11	**VANG**	Angle between vectors	Data in V1 and V2	Angle in X-reg	*Ángel Martin*
12	**VCHS**	Negative vector	Vector in V1	Negative vector in V1	*Ángel Martin*
13	**VENTER^**	Enters V1 into V2 level	Vector in V1	Pushes V1 into V2	*Ángel Martin*
14	**VIEWV**	Views V1 vector coordinates	Vector in V1	Shows X:, Y:, Z: coordinates	*Ángel Martin*
15	**VINV**	Vector Inverse	Vector in V2	Inverse vector in V1	*Ángel Martin*
16	**VMOD**	Vector Modulus	Vector in V2	Modulus in X-reg	*Ángel Martin*
17	**VNORM**	Vector Norm	Vector in V2	Norm in X-reg	*Ángel Martin*
18	**VUNIT**	Unitary Vector	Vector in V2	V/|v| in V1	*Ángel Martin*
19	**V=0?**	Conditional V1=0?	Vector in V2	YES/NO, Do if True.	*Ángel Martin*

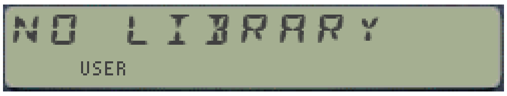
20	V=A?	Conditional V1=V2?	Data in V1 and V2	YES/NO, Do if True.	*Ángel Martin*
21	V#0?	Conditional V1#0?	Vector in V2	YES/NO, Do if True.	*Ángel Martin*
22	V#A?	Conditional V1#V2?	Data in V1 and V2	YES/NO, Do if True.	*Ángel Martin*
23	-AUX FNS	*High Rollers Game*	*none*	*The game begins…*	*Ross Cooling*
24	^MROW	Input Row	"OP1" in Alpha, row# in x	Prompts for Row	*Ángel Martin*
25	ABSP	Alpha Back Space	Text in Alpha	Last chr deleted	*W&W GmbH*
26	AIP	Appends integer part	x in X	INT(x) appended to Alpha	*Ángel Martin*
27	ASWAP	Alpha Swap	"A,B" in Alpha	"B,A" in Alpha	*Ángel Martin*
28	CLAC	CLA from Comma	Text in Alpha	Removed from left to comma	*W&W GmbH*
29	DTC	Delete Tiny Coefficients	Cntl word in X	Tiny coeffs deleted	*Ángel Martin*
30	DOTN	N-dimensional Dot product	cnt'l words in Y,X	cnt'l word result in X	*JM Baillard*
31	I*J	Multiplies indexes	iii.jjj in X	Product i*j in X	*Ángel Martin*
32	I#J?	Are indexes the same?	iii,jjj in X	YES/NO, Do if True.	*Ángel Martin*
33	MTR	Matrix Subroutine	Under program control	Under program control	*HP Co.*
34	OCX	Output x-th column	"OP1" in Alpha, Col# in X	Shows Col elements	*Ángel Martin*
35	ORX	Output x-th row	"OP1" in Alpha, Row# in X	Shows Row elements	*Ángel Martin*
36	PF>X	Prime Factors to Number	Matrix w/ Prime Facts in XMEM	Restores the original argument	*Ángel Martin*
37	PL2	2D Point-to-Line distance	Coordinates in Stack	Distance in X-reg	*Ángel Martin*
38	PP2	2D Point-to-Point distance	Coordinates in Stack	Distance in X-reg	*Ángel Martin*
39	ST<>A	Swaps Alpha/Stack	V1 in Stack, V2 in Alpha	V2 in Stack, V1 in Alpha	*Ángel Martin*
40	TOTNT	Totient Function	Argument in X	Result in X	*Ángel Martin*
41	-REV	*Revision Message*	*none*	*Splash screen (!)*	*Nelson Crowle*
42	?IRR	Irreducibility Criteria	Control word in X	Stops If irreducible, R04 has #	*Ángel Martin*
43	CROUT	Outputs Cubic Roots	Roots in Stack	Shows messages	*Ángel Martin*
44	CT	Subroutine for "TR"	Under program control	See Advantage’s manual	*HP Co.*
45	dPL	Polynomial 1st. derivative	Cntl word in Y, x in X	P’(x) in X, argument in L	*Ángel Martin*
46	dPL2	Polynomial 2nd. derivative	Cntl word in Y, x in X	P”(x) in X, argument in L	*Ángel Martin*
47	HMT	Hermite Polynomials	N in Y, x in X	H(n, x) in X, argument in L	*Ángel Martin*
48	ITPL	Polynomial Primitive	Cntl word in Y, x in X	Integralin X, argument in L	*Ángel Martin*
49	LAG	Laguerre Polynomials	N in Y, x in X	H(n, x) in X, argument in L	*Ángel Martin*
50	LANX	Associated Laguerre Pols.	A in Y, N in Y, x in X	Result in X	*Ángel Martin*
51	LEG	Legendre Polynomials	N in Y, x in X	H(n, x) in X, argument in L	*Ángel Martin*
52	P4	Auxiliary Code for QUART	Coeffs. In stack, UF 00 status	Coeffs for CROOT	*Ángel Martin*
53	PSWP	Polynomial Swap	Control word in Y, new rg in X	Coeeficients Swapped	*JM Baillard*
54	CAT+ _	Sub-function Catalog	Has hot keys	Lists/Launches functions	*Ángel Martin*
55	CMINOR	Complex Matrix Minor	MNAME in ALPHA, (I,j) in X	Result in ALPHA and (Y,X)	*Ángel Martin*
56	CMTRP	Complex Matrix Transpose	MNAME in ALPHA	Transposed Matrix	*Ángel Martin*
57	M^1/X	Matrix inverse Power	X in X, MNAme in ALPHA	Result in Matrix if convergence	*Ángel Martin*
58	MRND	Random Matrix	Dim in X, MName in ALPHA	Random Elements	*Ángel Martin*
59	MSQ2	2x2 Matrix Square Root	MNAME in Alpha	Square root replaces Matrix	*Ángel Martin*
60	MZDG	Matrix Zero Diagonal	“OP1” in Alpha	Clears the Diagonal Elements	*Ángel Martin*
61	PSCAL	Pascal Matrix	"OP1" in Alpha	Pascal elements	*Ángel Martin*
62	RMCAT	R-Matrix Catalog	Has hot keys: D,H,SST,SHF, R/S	Lists R-Matrices in RAM	*Ángel Martin*
63	YMCAT	Y-Matrix Catalog	Has hot keys: D,H,SST,SHF, R/S	Lists R-Matrices in RAM	*Ángel Martin*
64	@+	Append Element	Under program control	Used by MCEDIT and IMR	*Ángel Martin*
65	AIM	Anti-Identity Matrix	Matrix name in ALPHA	Unitary with zero diagonal	*Ángel Martin*
66	EQT	Display CurveFit Equation	Eq# in R00 (1-16)	Eq. displayed in Alpha	*Ángel Martin*
67	EV2X2	2x2 Matrix Eigenvalues	Matrix Name in ALPHA	Eigenvalues in Y,X	*Ángel Martin*

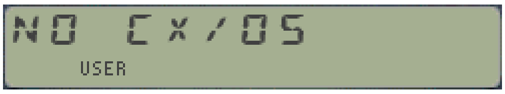
The function groups are distributed in both lower and upper pages, as follows:

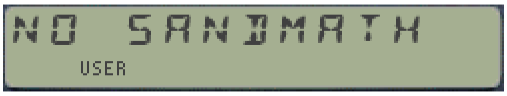
- The lower page contains the general intro section plus the CCD Matrix set. Very much like the lower page of Advantage Pac minus the digital functions. It also hosts the auxiliary FAT, a necessity in order to have the subfunctions available to the FOCAL programs within the ROM. The second bank in the lower page contains the 3D Vector calculator functions, the execution tables for the three launchers, the code for the orthogonal polynomials, the Matrix CATAlogs, and and the **EQT** function. Also there are the "return" snippets to support the Advantage-style bank-switching calls to the upper page.

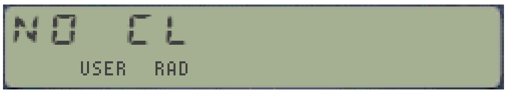
- The upper page has the Advanced Matrix and Polynomial sections. Basically all new and additional to the Advantage Pac. Finally, The second bank in the upper page is practically identical to that in the Advantage, with a few changes made after removing the Digital functions as well. It mostly contains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

The SandMatrix checks for the presence of its dependencies, i.e. The Library#4 and the SandMath. Note that if the SandMath module is not plugged in the calculator a warning message is shown every time the calculator is switched on -- but not halting the polling points process, whereas if the Library#4 is missing or the machine is not a CX, the errors will halt it to avoid likely problems.


NO LIBRARY
USER


NO CX/OS
USER


NO SANDMATH
USER

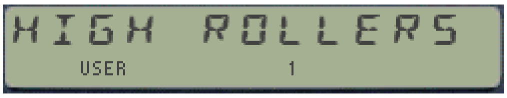

NO CL
USER RAD

Note that there's also a punctual check for the CL board – needed to ensure the Y-registers existence if a matrix is to be stored in the CL-RAM area. This check is only performed within the affected functions, and not at initialization.

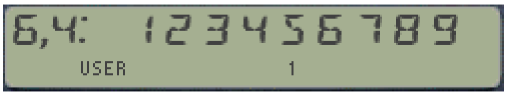
Summarizing, the SandMatrix is a complete algebra module not only with powerful Matrix capabilities (inherited and extended on its own), but also Vectors and Polynomial functions second to none in the HP-41 platform. Yet its FAT internal architecture and launcher menus should make its usage intuitive and easy to learn… hopefully you'd agree and have as much fun using it as I've had putting it together.

Xtra Bonus:- High Rollers Game.

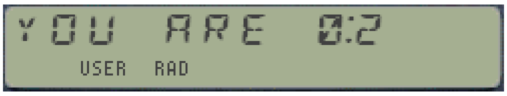
There is a gigantic Easter egg included in the SandMatirx – hidden somewhere there's a rendition of the High Rollers game, so you can relax in between hard-thinking sessions of math, really! There was simply too much available space in the "new" bank of the lower page to leave it unused, so this 500+ bytes MCODE rendition of the game (written by Ross Cooling, see PPCJ V14N2 p31) was begging to be included. As to how to access it… the discovery is part of the enjoyment :-) Hint: ever wondered about those section header functions?...


HIGH ROLLERS
USER 1

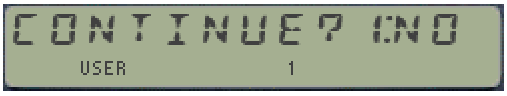
,


6,4: 123456789
USER 1

Choose any combination from the available digits on the right which sum matches the target on the left, repeating until there's no more digits left (YOU WIN) or there aren't possible combinations (YOU LOSE). Use R/S to proceed, back-arrow to delete digits. The game will ask you to try again and will keep the count of the scores.


YOU ARE 0:2
USER RAD

,


CONTINUE? !:NO
USER 1

Note: Make sure that revision "R58" (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. SNDMATRX"4

2.1. Alpha String Management sub-functions

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 a primary 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 that module as another dependency (even if it's just a 4k footprint for its 4 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	**ST<>A**	Exchanges Alpha and Stack	Values in Stack and Alpha registers

Note that already in revision "M" all these functions weren moved to the secondary FAT, and therefore you need to use the **ΣV$** or **ΣV#** launchers to execute them, or to enter them in a FOCAL program. See the sub-function table at the introduction section for details on their reference indexes.

- **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.

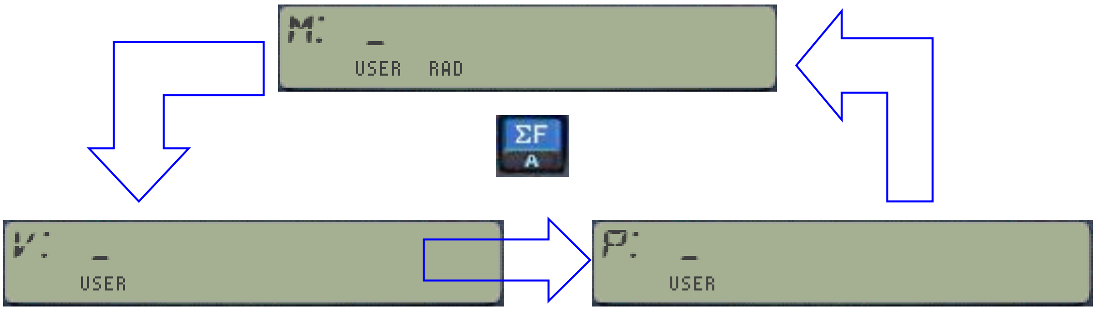
- **CLAC** deletes the contents of ALPHA located to the right of a comma (i.e. after the comma but not including it). It is adapted from **CLA-** in the CCD Module.

- **ST<>A** simply exchanges the contents of the stack and the four Alpha registers {M,N,O,P}. This is different from **V<>A**, in that only the three stack registers are swapped. **V<>A** is used in 3D-vector operations where one of the operands is stored in Alpha.

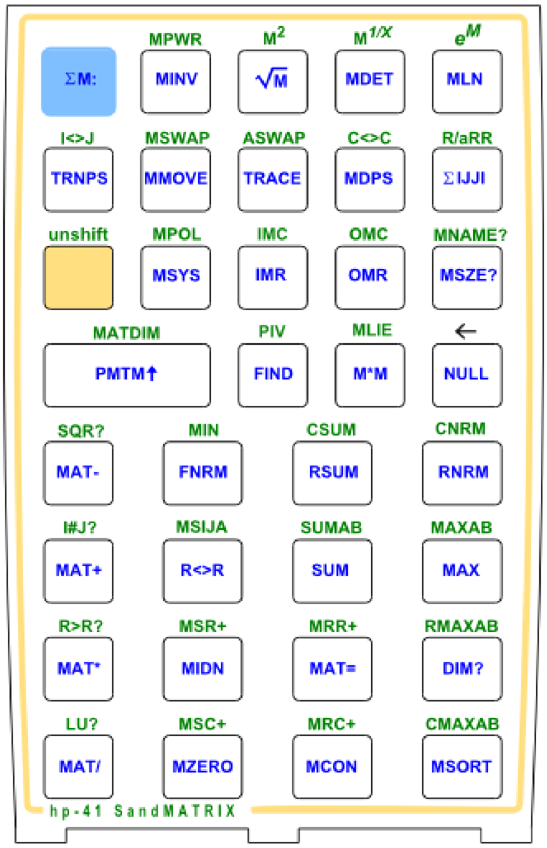
2.2. The Matrix function launcher and Keyboards.

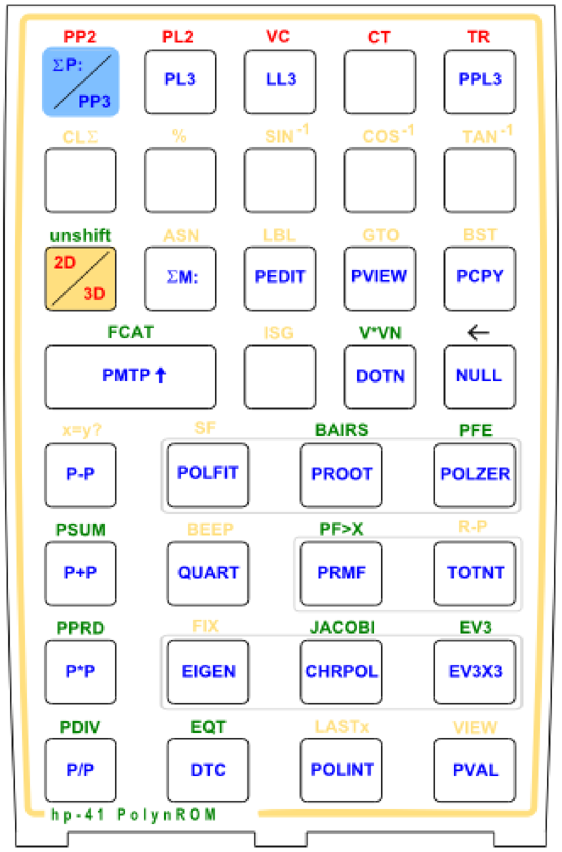
It has become customary in the author's modules to always use function launchers to enhance the user interface, as the most convenient way to save key assignments. As expected, there is a function launcher in the SandMatrix that groups many of the most frequently used matrix functions; plus a few others thrown to the mix. All functions can be accessed from the "**M: _**" prompt, there is no need to make any key assignment other that for the main launcher itself.

This becomes active when you execute "**ΣML**" - the function launcher in run mode. When executed, it presents the "**M:**" prompts and awaits your next selection. Pressing [**A**] in turn activates the Vector prompt **V:_**, and pressing it a second time the Polynomial prompt **P:_**.



The following pictures of the Matrix and Polynomial Overlays show the available functions in the dedicated keyboards. Note that those with blue names (on the keys) are called using the main keys (ie. unshifted), and those in green (above the keys) require pressing the [SHIFT] key in the corresponding prompt. It also shows the **M:** launcher assigned to the Σ+ key, as usual.





2.3. Functions in the Header section.

#	Function	Description	Input
1	**"MATRX"**	"Easy Matrix" Program	Driver for Major Matrix Ops.
2	**ΣML _**	Matrix Function Launcher	Prompts for function
3	**ΣDST _**	Distance Functions Launcher	Prompts "PP PL LL PPL"
4	**ΣV# _ _ _**	Sub-Function Launcher	Prompts for index
5	**ΣV$ _**	Sub-Function Launcher	Prompts for name

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.

2.3.1 The Matrix Program

This chapter describes the matrix program, **MATRX** - the menu-driven, 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.

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

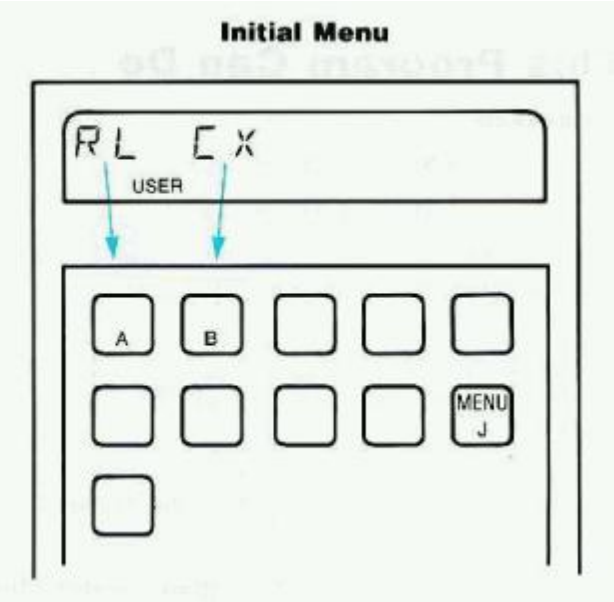
For such a matrix system, the **MATRX** program creates (dimensions) a square real and complex matrix, **A**, and a column matrix, **B.** You can then:

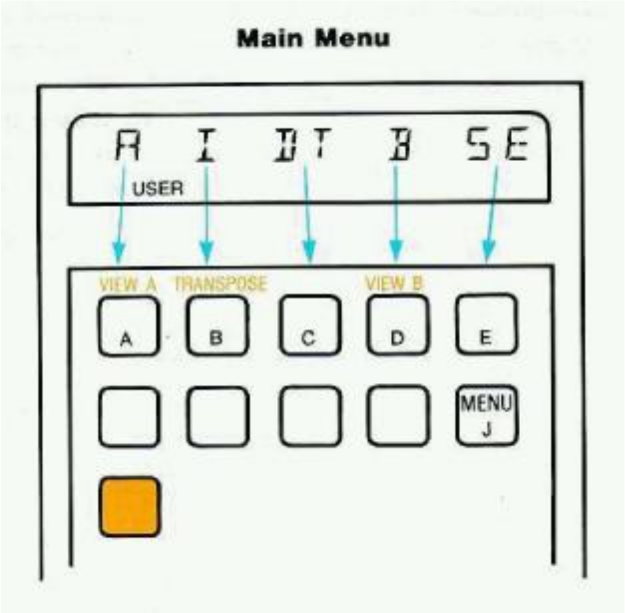
- Enter, change ('edit"), or just view elements in **A** and **B.**
- Invert **A.**
- Transpose **A** if **A** is real.
- Find the determinant of **A** if **A** is real.
- Solve the system of simultaneous equations by finding the solution to **AX = B.**

The size of your matrix is limited only by available memory (each real matrix requires one register plus 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 column 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 prompts 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 original 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-decomposition 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 Functions' for an explanation 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 manual for the HP-41) assigned to keys [**A**]-[**E**], [**J**] , [**a**], [**b**], and [**d**]. These local assignments are overridden by any User-key assignments you might have made to these same keys, thereby defeating this program. Therefore be sure to clear any existing User-key assignments of these keys 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 “**MATRX**”	“RL CX”	Starts the MATRX 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)
[**D**] (B)	“ 1:1=b11?“	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	"RE 1:1= ?"	Enters the editor and displays old value
1, R/S	"IM 1:1= ?"	ditto for the imaginary part
2, R/S	"RE 1:2= ?"	
3, R/S	"IM 1:2= ?"	
4, R/S	"RE:2:1= ?"	Wrong entry! Should be 3, not 4...
1,002, [**A**]	"RE 1:2= 3.000?"	Moves editor back to x_{12}
R/S	"IM 1:2= 4.000?"	The wrong imaginary part
3, R/S	"RE 2:1= ?"	Correct value is entered for y_{12}. Proceed
4, R/S	"IM 2:1= ?"	
5, R/S	"RE 2:2= ?"	
6, R/S	"IM 2:2= ?"	
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**]	"RE 1:1= -0.9663"	Viewing A^{-1}
R/S	"IM. 1,1= 0.1461"	
R/S	"RE. 1,2= 0.4382"	
R/S	"IM. 1,2= -0.1011"	
R.S	"RE.2,1= 0.6742"	
R/S	"IM.2,1= -0.0787"	
2.002, [**A**]	"RE 2:2= -0.2369"	Displays x_{22} + i y_{22}
R/S	"IM 2:2= -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, with the description of the individual matrix functions.

Note that if any of the element registers currently contains alpha data you'll get an ALPHA DATA error message during data entry. Simply clear the X register (using tha back arrow key) and continue with R/S.

See the program listing in next page for your reference.

```
*LBL "MATRX"
*LBL 00
 CF 02
 SF 27
 "RL CX"
 PROMPT
 GTO 00
*LBL B
 SF 02
*LBL A
 ?ORDER
 STOP
 ABS
 INT
 ENTER^
 FS? 02
 +
 STO 00
 X^2
  E
 +
 SIZE?
 X<>Y
 X>Y?
 PSIZE
  E
 E3/E+
 RCL 00
 *
 "R"
 MATDIM
 GTO 01
 RTN
*LBL “MTR”
*LBL J
*LBL 01
 CF 08
 SF 27
 " A I DT B SE"
 AVIEW
 "R"
 STOP
 GTO 01
*LBL 03
 MDET
 "DET="
 ARCL X
 PROMPT
 GTO 01
*LBL a
 SF 08
*LBL A
 GTO 14
*LBL B
 MINV
 GTO 01
*LBL C
 FC? 02
 GTO 03
 XROM "CMDET"
 ΣF# 032
 (ZOUT)
 STOP
 GTO 01
*LBL E
 XEQ 02
 ASTO X
 "R, "
 ARCL X
 MSYS
 GTO 01
*LBL b
 FC? 02
 TRNPS
 FC? 02
 GTO 01
 ΣV# 056
 (CMTRP)
 GTO 01
*LBL d
 SF 08
*LBL D
 XEQ 02
*LBL 14
 SF 07
 FS? 02
 XROM "CMEDIT"
 XROM "MEDIT"
*LBL 02
 DIM?
 INT
 ENTER^
 X^2
  E
 +
 AINT
 RDN
 SF 25
 MATDIM
 FS?C 25
RTN
R^
+
 E
+
PSIZE
GTO J
END
```

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

Use the Alpha register to specify matrix names. When specifying more than one name (as parameters for certain functions), separate them with commas. 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 maximum square matrix size of 17x17 is possible in this modality.

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. A maximum square matrix size of 24x24 is possible in this modality.

For the SY-41CL there's a third option available that uses the *expanded memory* zone, which provides a storage space for additional 3,072 data registers (aka. the Y-Registers). To store your matrix in that area simply name it **Yxxx**, with the same conventions given for the standard main memory case. For example, using **Y525** will store the matrix elements starting at the 526th expanded register. Note that the expanded memory area allows for larger sizes – up to 55 x 55 square matrices !

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 t*he 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. *Displaced elements are zeroed if an existing matrix is being enlarged.*

- **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 and expanded Y-memory occupies (m x n) + 1 data storage registers, one register being used as a status header. A complex matrix uses (2m x 2n) + 1 registers, where m is the number of rows and n is the number of columns in the complex matrix.

A matrix in extended X-memory has a file length of (m x n). (2m x 2n for a complex matrix). Two more X-mem registers are used for the header and the file name. 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 **MZERO** - or **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 :

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

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 bounds 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.

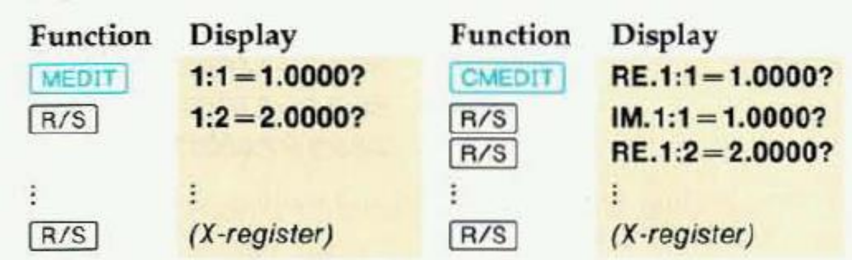
Existing matrices in extended memory cannot be redimensioned to completely fill extended memory. The maximum allowable size of a redimensioned matrix is one register less than the currently available extended memory. A new matrix can, however, be dimensioned to completely fill available extended memory.

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 part 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:

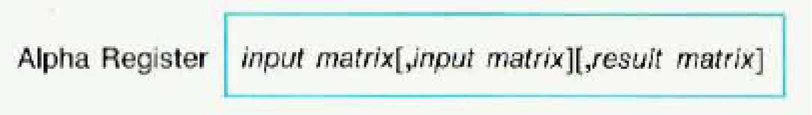
- Press [**J**].
- Try to access a nonexistent element. For instance, in a 4 x 4 matrix, press 5 [**A**].

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 operation, the calculator must be given the identities 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 arithmetic 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

Program Listings.-

```
01*LBL "MEDIT"
02 SF 27
03 0
04 MSIJA
05*LBL A
06 CF 10
07 SF 25
08 MSIJ
09 FC?C 25
10 GTO 05
11*LBL 00
12 AOFF
13 MRIJ
14 FS? 10
15 GTO 05
16 CLA
17 MPT
18 MR
19 ARCL X
20 FS? 08
21 GTO 08
22 >"?"
23 CF 23
24 PROMPT
25 MR
26 X<>Y
27 X#Y?
28 MS
29 ASTO X
30 FS? 23
31 MS
32*LBL 09
33 J+
34 GTO 00
35*LBL 08
36 PROMPT
37 GTO 09
38*LBL J
39 GTO J
40  END
```

```
01*LBL "CMEDIT"
02 SF 27
03 CF 06
04*LBL 06
05 DIM?
06 ODD?
07 GTO 00
08 FRC
09 I<>J
10 X=1?
11 GTO 00
12 EVEN?
13 GTO 00
14 "NOT CPX"
15 PROMPT
16 MNAME?
17 GTO 06
18 SF 06
19 CF 10
20 CLST
21*LBL A
22 ST+ X
23  E
24 E3/E+
25 -
26 SF 25
27 MSIJ
28 FC?C 25
29 GTO J
30*LBL 01
31 FS? 10
32 GTO J
33 XEQ 03
34 "RE."
35 XEQ 05
36 FS? 08
37 GTO 08
38 PROMPT
39 MS
40 FS? 06
41 GTO 08
42 J+
43 I+
44 MS
45 J-
46 I-
47*LBL 08
48 SF 21
49 FS? 08
50 AVIEW
51 XEQ 03
52 "IM."
53 I+
54 XEQ 05
55 FS? 08
56 GTO 09
57 PROMPT
58 MSR+
59 FS? 06
60 GTO 10
61 I-
62 CHS
63 MSC+
64 GTO 10
65*LBL 09
66 AVIEW
67 J+
68 FS? 06
69 GTO 01
70*LBL 10
71 FS? 10
72 GTO J
73 FC? 06
74 J+
75 FC? 09
76 I-
77 GTO 01
78*LBL 03
79 MRIJ
80  E
81 E3/E+
82 +
83 2
84 /
85 RTN
86*LBL 05
87 ΣV#
88 64
89 FC? 08
90 >"?"
91 RTN
92*LBL J
93 FS? 07
94 GTO 01
95 MNAME?
96 END
```

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 – uses current matrix pointer
6	**I-**	Decrements iii pointer by one	None – uses current matrix pointer
7	**J+**	Increments jjj pointer by one	None – uses current matrix pointer
8	**J-**	Decrements jjj pointer by one	None – uses current matrix pointer

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).

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

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.

Another, more effective way to enter the element values involves 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:

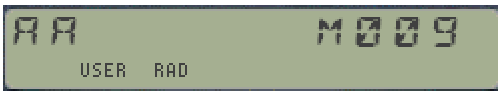
Keystrokes	**Display**	**Comments**
ALPHA, "ABC", ALPHA		
2.003, XEQ "**MATDIM**"	2.003	Dimensions matrix ABC in X-Mem.
XEQ "**PMTM**"	"R 1:"	prompts to enter the first row
5, ENTER^, 6, ENTER^, 7, R/S	"R 2:"	prompts for the second row
8, ENTER^, 9, ENTER^, 10, R/S		done!

Function **PMTM** will be covered later in the manual, as part of the new functions section.

2.2.3.- Matrix Catalogs. Where is my stuff? { RMCAT , YMCAT }

The ability to use both X-Mem and standard data registers was a design crtiteria in the Array functions of the CCD Module. The reason was clearly to make the functionality available to plain HP-41C machines, without the X-Functions and X-Memory. This was proven to be the right decision, albeit it made the internal routines more complex since they needed to support two memory areas.

You can use CAT'4 (or EMDIR) to list all files in X-Mem, including the Matrix files of course. With the AMC_OS/X module plugged in the "M" type is shown in the LCD, indicating the Matrix type.



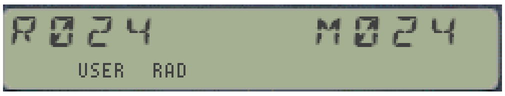
AA M009
USER RAD

Until now, there was no way to enumerate the matrices "configured" in the standard registers – with matrix names "**Rxxx**" as described in the previous sections of the manual. Even if these were assumed to be more volatile than the X-Mem counterparts, it would have been very helpful to have a matrix catalog of some sort.

Support for the CL expanded memory adds another dimension to the same feature, and given the vast increase of available memory – and the fact that it is much less labile - it becomes even more important to know which matrices are already configured in the standard and Y-Registers.

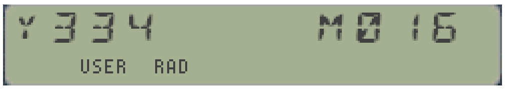
New Matrix Catalogs

The SandMatrix includes two sub-functions to enumerate the matrices configured in the standard and CL register areas; **RMCAT** and **YMCAT** respectively. No input data is required, and the enumeration will sequentially show all matrices present in the corresponding zone. Like in the EMDIR case, both the matrix "name" and size will be shown in the display:



R024 M024
USER RAD

or:

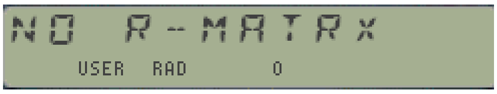


Y334 M016
USER RAD

The listing can be stopped and resumed using the R/S key. Whilst stopped you can use the SST and BST keys to advance or backtrack the listing. Other hot keys are available to delete the matrix, enter its name to ALPHA, or decode the header register:

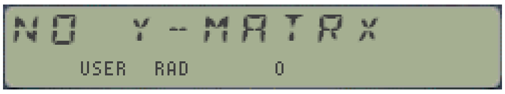
R/S	Switches between single-step or automated enumeration
[SHIFT]	reverse or direct enumeration
SST	moves to next matrix if present
BST	moves to previous matrix if present
[**H**]	Decodes the header register
[**D**]	Deletes the matrix (asks for confirmation first)
ENTER^	Enters the matrix name in ALPHA, ready to use all matrix functions.
Back Arrow	Exits the enumeration

Finally, if no matrices are found in that memory area the function will put up one of the info messages shown below, depending on the case:



NO R-MATRX
USER RAD 0

or:



NO Y-MATRX
USER RAD 0

I trust you'll agree this pair of functions is a welcome addition to the SandMatrix, well worth the price of admission and the extra code added to the project.

Appendix. Harmonic Determinants. { DNN , DN }

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

Consider the determinant D(N) defined as follows:

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

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

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

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

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

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

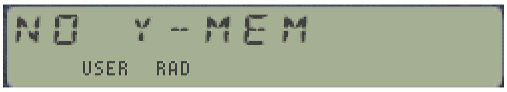
```
01 LBL "DDN"
02 RCL X
03 E3
04 /
05 +
06 "Y" - matrix will start at RY-001
07 MATDIM
08 1
09 MCON
10 CLX
11 MSIJA
12 2
13 LBL 00
14 E
15 +
16 MSC+
17 SF 25
18 J+
19 FS?C 25
20 GTO 00
21 MDET
22 END
```

```
01 LBL "DN"
02 E
03 +
04 HARM
05 LASTX
06 FACT
07 *
08 END
```

And the table below shows the results from each approach:

N	D(N)	Time (@Turbo50)	Formula
11	1,486,442,880.0	1.8 sec	1,486,442,880.0
13	2.83465647**2** E11	2.01 sec	2.834656474 E11
30	3.311538747 E34	11 sec	3.311538746 E34
40	1.439439902 E50	1 min 20 sec	1.439439902 E50
50	3.278748200 E75	2 min 30 sec	3.278748199 E75

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



NO Y-MEM

USER RAD

2.3. M-FUNCTIONS

-CCD MTRX

USER

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 linear 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 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.

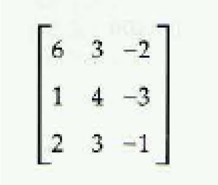
I recall the awe with which we used to run **MINV** and the other functions: just a single keystroke doing the same as all those intricate FOCAL programs did using Gaussian algorithms, element pivoting and row simplification... simply amazing back then. It was the ultimate Matrix function set, pretty 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.

Note: 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 calculations 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 wanted 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.

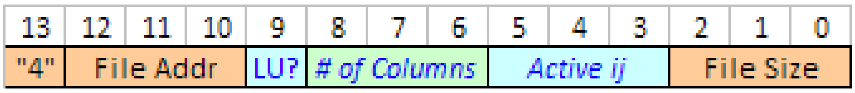
Editing *any* element of the matrix will clear the LU-flag in the status register, which makes the matrix unrecognizable to the program. Because of this, if you edit one element, you must edit them all if you wish to use the matrix again. Note that the matrix will no longer be in LU-decomposed form after 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, but in a program will halt the execution if true (i.e. it's 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**) "recognize" 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 operations. 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

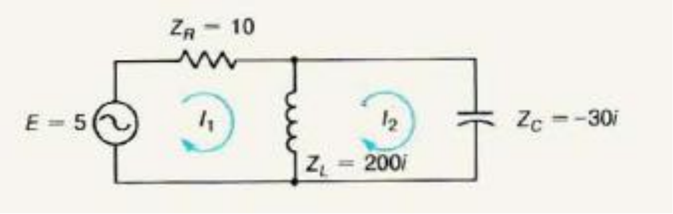
Most matrix functions do not operate meaningfully on complex matrices: since they don't recognize the different parts of a complex number as a single number, the results returned are not what you would expect for complex entries.

Valid Complex Operations. Certain matrix functions work equally well with real and complex functions. Both the input and result matrices must be complex. These functions are:

- **MSYS** Solving simultaneous equations
- **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

Example 2.

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.

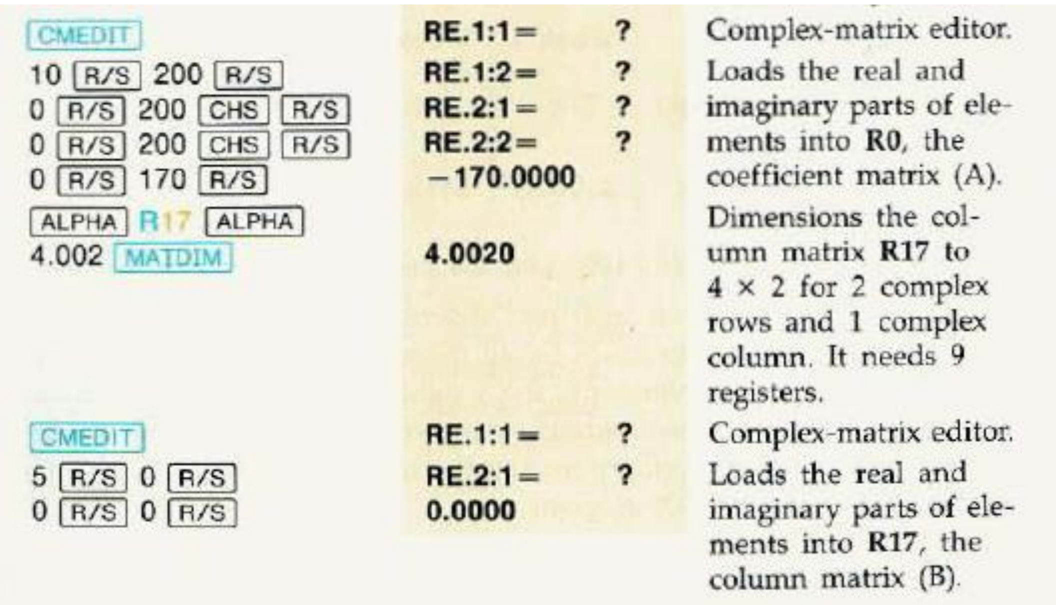
The main thing to sort out in this example is the dimension of the matrices involved. The coefficients matrix **A** is a 2 x 2 complex matrix, thus as per the previous paragraphs we will need (4x4 +1) = 17 registers. The independent terms matrix **B** is a 2 x 1 complex matrix, thus will need (4x2 +1)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

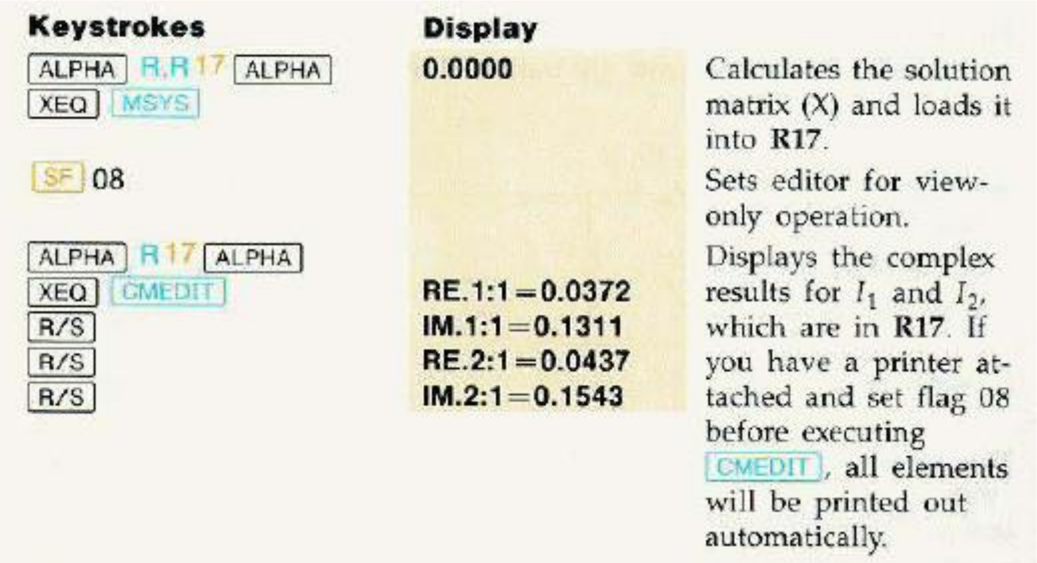
ALPHA, "R", ALPHA 4.004, XEQ "**MATDIM**"	ALPHA, "R17", ALPHA 4.002, XEQ "**MATDIM**"

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	0.0000	Calculates the solution matrix (X) and loads it into **R17**.
XEQ MSYS		
SF 08		Sets editor for view-only operation.
ALPHA R17 ALPHA		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.
XEQ CMEDIT	RE.1:1=0.0372	
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.

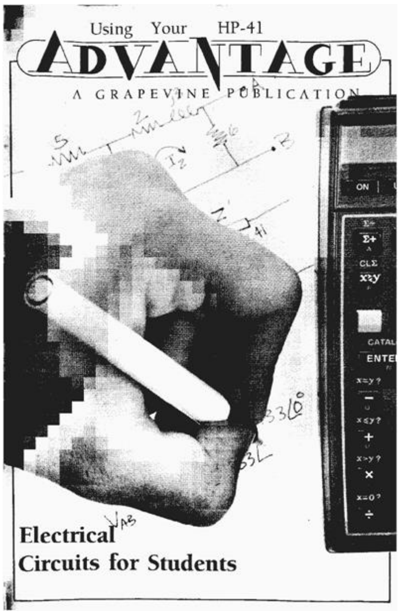
The program is documented in its dedicated Grapevine booklet, available at:

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 Science 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 for three 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 constant 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 sought 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

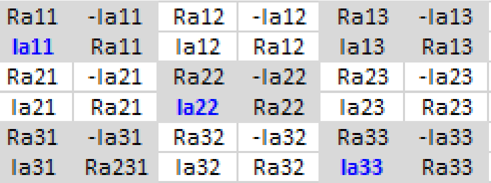
Note: using **OMR** (or **OMC**) to output the elements of the matrix B you can see how the results are all *integer values* – which speaks of the accuracy of the internal operations, taking advantage of the 13-digit math routines available in the OS for MCODE.

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.

Appendix. Complex Matrix Determinants. { CMTRC , CMDET }

The programs below are a first-pass successful attempt at calculating Complex Matrix determinants up to order 4. The Complex Matrix is to be stored using the SandMatrix convention - which is identical to the HP-41 Advantage's. With this convention each complex number is represented by four elements in the complex matrix - refer to the manuals for details.



Ra11	-Ia11	Ra12	-Ia12	Ra13	-Ia13
Ia11	Ra11	Ia12	Ra12	Ia13	Ra13
Ra21	-Ia21	Ra22	-Ia22	Ra23	-Ia23
Ia21	Ra21	Ia22	Ra22	Ia23	Ra23
Ra31	-Ia31	Ra32	-Ia32	Ra33	-Ia33
Ia31	Ra231	Ia32	Ra32	Ia33	Ra33

example for 3x3 case

The SandMatrix comes well-equipped with routines to calculate the trace and integer powers of a matrix (**MTRACE**, **M^2** and **MPWR**), therefore it lends itself rather nicely to the direct formulas using those elements, as described at: https://en.wikipedia.org/wiki/Determinant

$$\det(A) = \left((\operatorname{tr} A)^2 - \operatorname{tr}(A^2)\right)/2,$$

$$\det(A) = \left((\operatorname{tr} A)^3 - 3\operatorname{tr} A\ \operatorname{tr}(A^2) + 2\operatorname{tr}(A^3)\right)/6,$$

$$\det(A) = \left((\operatorname{tr} A)^4 - 6\operatorname{tr}(A^2)(\operatorname{tr} A)^2 + 3(\operatorname{tr}(A^2))^2 + 8\operatorname{tr}(A^3)\ \operatorname{tr} A - 6\operatorname{tr}(A^4)\right)/24.$$

The complex matrix won't be altered in any way, as all operations are made on a scratch copy. It can be stored in X-Mem, CL_Y-Mem, or standard data registers area. The easiest way to enter the matrix is by using the **CMEDIT** routine - which expects the matrix name in ALPHA. It expects the matrix already created, using 2n x 2n as dimension - with "n" being the order.

If you place it in the standard registers area, be aware that data registers R00, R01 are used by the routine **MPWR** for scratch. Additionally, data register R02 is used to store the Matrix Name (thus it can't exceed 6 characters).

As you can see there are numerous 41Z functions - used for the complex arithmetic using the Complex Stack. This has the additional advantage that doesn't require additional data registers, be that standard or CL Y-RAM.

Example.- Calculate the determinant of the 4x4 Complex Matrix:

$$\begin{vmatrix} 1+i & 2+2i & 3+3i & 4+4i \\ 0 & 1 & -3-3i & -4-4i \\ -1+i & 1-i & 1 & i \\ -i & -1+i & 1 & 0 \end{vmatrix}$$

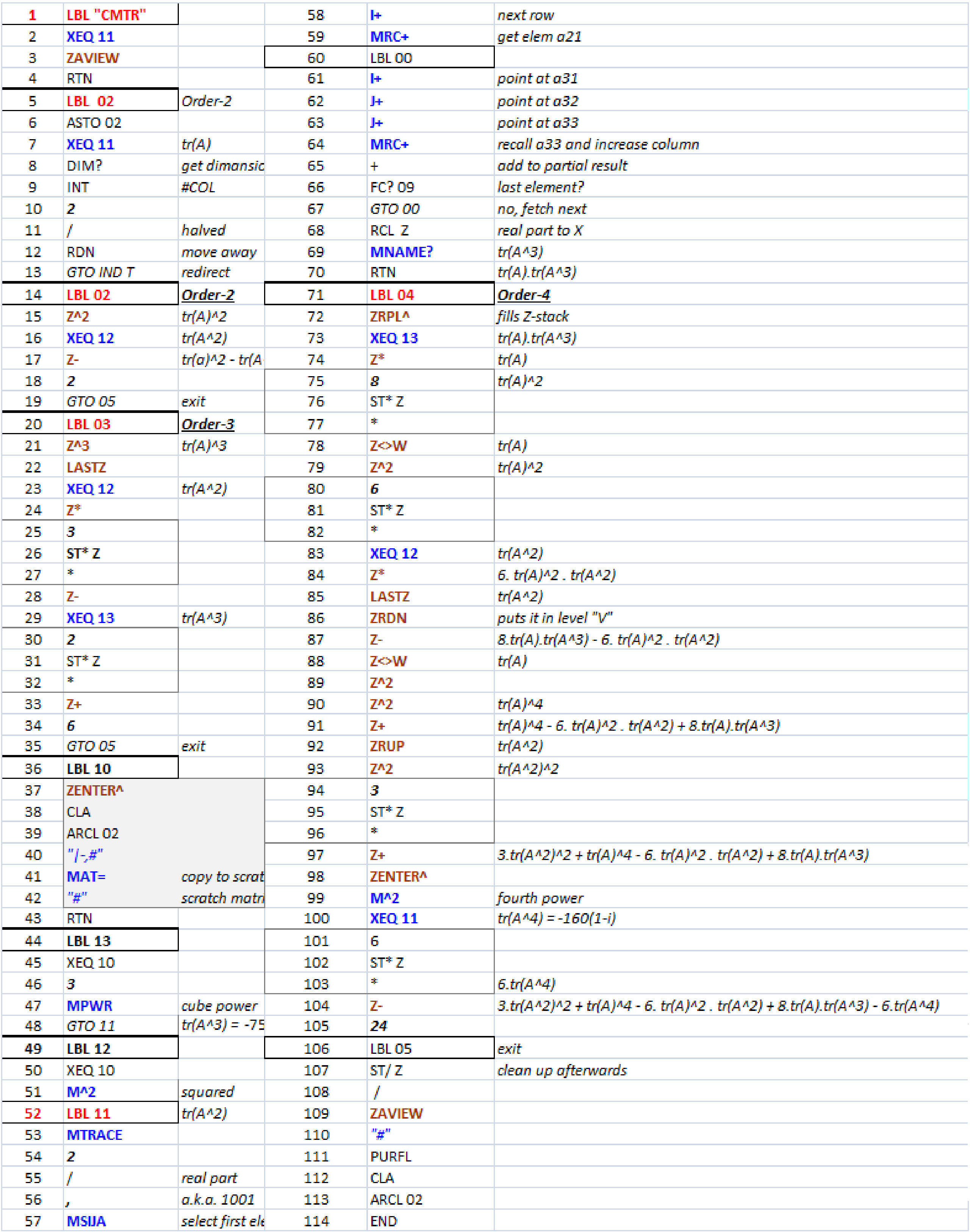
Solution: det = -62-8i

https://www.wolframalpha.com/input/?i=det(%7B%7B1%2Bi,+2%2B2i,+3%2B3i,+4%2B4i%7D,+%7B0,+1,+-3-3i,+-4-4i%7D,+%7B-1%2Bi,+1-i,+1,+i%7D,+%7B-i,+-1%2Bi,+1,+0%7D%7D)

The program is slow in non-turbo settings- there are lots of moving pars behind the scene, despite the straight-forward program listing. Using TURBO_50 the 4x4 determinant is obtained in 5 seconds approx.

The accuracy for integer matrices holds up nicely, giving exact integer real and imaginary parts in the solution.

Program Listing.



1	LBL "CMTR"		58	I+	next row
2	XEQ 11		59	MRC+	get elem a21
3	ZAVIEW		60	LBL 00	
4	RTN		61	I+	point at a31
5	LBL 02	Order-2	62	J+	point at a32
6	ASTO 02		63	J+	point at a33
7	XEQ 11	tr(A)	64	MRC+	recall a33 and increase column
8	DIM?	get dimansic	65	+	add to partial result
9	INT	#COL	66	FC? 09	last element?
10	2		67	GTO 00	no, fetch next
11	/	halved	68	RCL Z	real part to X
12	RDN	move away	69	MNAME?	tr(A^3)
13	GTO IND T	redirect	70	RTN	tr(A).tr(A^3)
14	LBL 02	Order-2	71	LBL 04	Order-4
15	Z^2	tr(A)^2	72	ZRPL^	fills Z-stack
16	XEQ 12	tr(A^2)	73	XEQ 13	tr(A).tr(A^3)
17	Z-	tr(a)^2 - tr(A	74	Z*	tr(A)
18	2		75	8	tr(A)^2
19	GTO 05	exit	76	ST* Z	
20	LBL 03	Order-3	77	*	
21	Z^3	tr(A)^3	78	Z<>W	tr(A)
22	LASTZ		79	Z^2	tr(A)^2
23	XEQ 12	tr(A^2)	80	6	
24	Z*		81	ST* Z	
25	3		82	*	
26	ST* Z		83	XEQ 12	tr(A^2)
27	*		84	Z*	6. tr(A)^2 . tr(A^2)
28	Z-		85	LASTZ	tr(A^2)
29	XEQ 13	tr(A^3)	86	ZRDN	puts it in level "V"
30	2		87	Z-	8.tr(A).tr(A^3) - 6. tr(A)^2 . tr(A^2)
31	ST* Z		88	Z<>W	tr(A)
32	*		89	Z^2	
33	Z+		90	Z^2	tr(A)^4
34	6		91	Z+	tr(A)^4 - 6. tr(A)^2 . tr(A^2) + 8.tr(A).tr(A^3)
35	GTO 05	exit	92	ZRUP	tr(A^2)
36	LBL 10		93	Z^2	tr(A^2)^2
37	ZENTER^		94	3	
38	CLA		95	ST* Z	
39	ARCL 02		96	*	
40	"|-,#"		97	Z+	3.tr(A^2)^2 + tr(A)^4 - 6. tr(A)^2 . tr(A^2) + 8.tr(A).tr(A^3)
41	MAT=	copy to scrat	98	ZENTER^	
42	"#"	scratch matri	99	M^2	fourth power
43	RTN		100	XEQ 11	tr(A^4) = -160(1-i)
44	LBL 13		101	6	
45	XEQ 10		102	ST* Z	
46	3		103	*	6.tr(A^4)
47	MPWR	cube power	104	Z-	3.tr(A^2)^2 + tr(A)^4 - 6. tr(A)^2 . tr(A^2) + 8.tr(A).tr(A^3) - 6.tr(A^4)
48	GTO 11	tr(A^3) = -75	105	24	
49	LBL 12		106	LBL 05	exit
50	XEQ 10		107	ST/ Z	clean up afterwards
51	M^2	squared	108	/	
52	LBL 11	tr(A^2)	109	ZAVIEW	
53	MTRACE		110	"#"	
54	2		111	PURFL	
55	/	real part	112	CLA	
56	,	a.k.a. 1001	113	ARCL 02	
57	MSIJA	select first ele	114	END	

Update: Complex (n x n) Matrix Determinants. { CDMET }

What follows is a more powerful approach to the calculation of the determinant without restriction of the complex matrix dimension, other than the available memory of course. This new approach is not only more powerful but it also has the advantages of less execution time and shorter program length, a winner by all accounts.

The beauty of this approach is that we’ll put the characteristic polynomials coefficients to work bigtime to obtain the matrix determinant (independent tern). So **CMDET** works uses an iterative method to calculate the coeficients, given the fact that the determinant is the last coefficient of said characteristic polynomial with an order-dependent changed sign:

$$P(h) = \sum_{k=0}^{n} c_{n-k}\, h^k \text{ , and: } \det(A) = (-1)^n.cn \text{ ,}$$

i.e. the coefficient of the independent term. The program will leave the original matrix unchanged and its name in ALPHA upon completion.

It’s worth mentioning that the same iterative approach is also used by the **CHRPOL** routine to calculate the Characteristic Polynomial and its roots (i.e. the Matrix Eigenvalues); although in the CHRPOL implementation the coefficient of the independent term is obtained via **MDET** – which is pretty sensible for the real matrix case of course.

Note: You can refer to the Complex Matrix ROM manual for an extended version of this routine that exploits this method to its full extent to also calculates the inverse of a complex n x n matrix – along with the complete complex characteristic polynomial of course.

Register, Flags, etc.

Data registers {R00 – R01} are used in main memory.

Only user flag 7 is used to signal when the routine is called from the main MATRX program..

Two auxiliary matrices are needed, each of the same size of the original one – therefore the maximum complex matrix size is n= 7 in X-Memory, (for a total of 588 X-mem registers), and n= 15 in the CL Y-Memory (for a total of 2,700 Y-registers). This limitation stems from the MCODE function **M*M**, which cannot have the result matrix equal to any of the operands (i.e. it’s not an in-place multiplication). But I’m not complaining, *since it does support complex matrices* as well as real ones!

Acknowledgment- Many thanks to Valentín Albillo for piquing my curiosity with his powerful and elegant implementation of the same algorithms on the HP-71, described with numerous examples and insights in the article posted here:

HP Article VA047 - Boldly Going -Eigenvalues and Friends

Program listing.

```
01*LBL "CMDET"
02 FC?C 07        ` called from MATRX?
03 GTO 10         ` yes, divert
04 "R,="          `no, uses data regs
05 MAT=
06 MNAME?
07*LBL 10
08 ASTO 01
09 >",P"          'aux. matrix
10 MAT=           `make equal
11 DIM?
12 "#"            `aux. matrix
13 MATDIM
14 FRC
15 2
16 /
17 LASTX
18 +
19 STO 00
20 CF 21
21*LBL 00
22 VIEW 00
23 XEQ 01
24 "#"
25 MIDN
26 "X,#,#"
27 MAT*
28 X<>Y
29 "#"
30 .
31 XEQ 02
32 "P,#,#"
33 MAT+
34 CLA
35 ARCL 01
36 >",#,P"
37 M*M
38 ISG 00
39 GTO 00
40 XEQ 01
41 RCL 00
42 ODD?
43 GTO 04
44 CLX
45 - E
46 ST* Z
47 ST* Y
48*LBL 04
49 RDN
50 "#"
51 PURFL
52 "P"
53 PURFL
54 ΣV#             `sub-routine?
55 GTO 04          `yes, skip
56 ΣF#
57 32
58*LBL 04
59 CLA
60 ARCL 01
61 RTN
62*LBL 02
63 2.002
64 +
65 SF 25
66 MSIJA
67 RDN
68 FC?C 25
69 RTN
70 I-
71 CHS
72 MSC+
73 J-
74 CHS
75 MSR+
76 MRIJ
77 GTO 02
78 RTN
79*LBL 01
80 "P"
81 CMTRC
82 RCL 00
83 INT
84  E
85 -
86 CHS
87 ST/ Z
88 /
89 END
```

Appendix. Complex Transposed Matrix. { CTRNP , ZTRNP }

Also missing from the original Advantage was a way to transpose a complex matrix; a fact that is duly corrected here with the routines below. Two solutions are offered, one following an element-based approach to illustrate the concept, and another using a block approach – faster and with fewer progam steps / byte count.

For a complex element pointer (i,j) the relationship with the individual marix pointers holding the four real and imaginary parts are as follows:

$$C(i,j) = [(2i-1, 2j-1) ; (2i-1, 2j) \\ (2i, 2j-1) ; (2i, 2j)]$$

- The element-based approach will simply do a cursory scan of the matrix, swapping the complex element with pointer (i.j) with its transposed one, (j,i), letting the real matrix pointers be determined by the relationships above. It also includes two subroutines for complex element recall and storage, **CMR** and **CMS** that can be used independently, entering with the complex pointer in X and the matrix name in ALPHA.

- The block-based approach takes advantage of the **MMOVE** function, applied to the four-element block as per the complex values storage convention. This requires just n x m iterations, whereas the previous approach needs 2n x 2m repeats of the single-element copying. Also no need to worry about the matrix pointers, MMOVE will take care of that for us.

- In either case one auxiliary matrix is required to perfrom the task, which will be purged on completion - leaving the (now transposed) matrix name in ALPHA. Neither of the two approaches requires data registers, but the first one uses flags 00 and 01.

Program listing – Block approach.

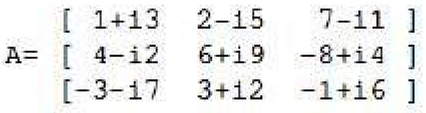
1	**LBL "ZTRNP"**	*FROM*	21	**MMOVE**	
2	"|-,#"	*FROM,TO*	22	X<>Y	*2i,2j*
3	**DIM?**		23	**MSIJA**	*position element in from matrix*
4	**I<>J**	*transpose dimension*	24	R^	*complex pointer (i,j)*
5	**ASWAP**	*TO,FROM*	25	**J+**	*test the location for bounds*
6	**MATDIM**		26	FS? 10	*out of matrix?*
7	**ASWAP**	*FROM,TO*	27	GTO 02	*yes, exit*
8	1,001	*complex pointer*	28	FS? 09	*out of rows?*
9	LBL 01	*prepare prameters*	29	INT	*yes, integer*
10	ENTER^	*i,j*	30	1.001	*offset factor*
11	ST+ X	*2i,2j*	31	FC? 09	*within rows?*
12	ENTER^		32	FRC	*yes, fractional*
13	**I<>J**	*2j,2i*	33	+	*update complex pointer*
14	1,001		34	*GTO 01*	do next block
15	ST- Z	*(2i-1),(2j-1) in Z*	35	LBL 02	
16	-	*(2j-1),(2I-1)*	36	**ASWAP**	
17	X<>Y		37	**MAT=**	*copies result & redims matrix*
18	R^	*i,j*	38	PURFL	*purges TO*
19	ST+ X	*2i,2j*	39	**MNAME?**	*FROM*
20	X<>Y		**40**	**END**	*85 bytes*

Program listing - Element-based approach

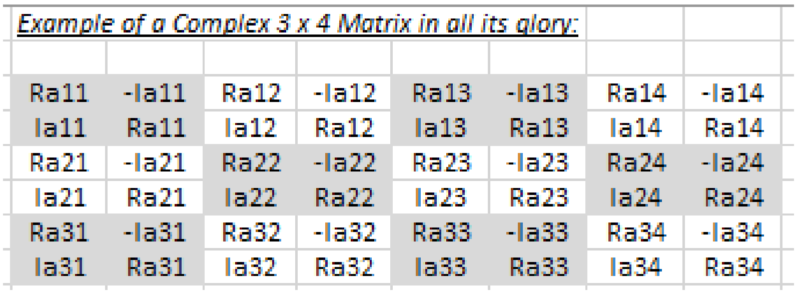
1	LBL "CTRNP"	*FROM*	32	*GTO 01*	*next complex element*
2	*"|-,#"*	*FROM,TO*	33	LBL "CMR"	*X holds (i,j)*
3	DIM?		34	ENTER^	*complex (i,j)*
4	I<>J	*transpose dimension*	35	ST+ X	*2i,2j*
5	ASWAP	*TO,FROM*	36	MSIJA	
6	MATDIM		37	RDN	
7	ASWAP	*FROM,TO*	38	MRR-	Re(aij)
8	1,001	*complex pointer*	39	MRR+	Im(aij)
9	CF 00		40	X<>Y	*leaves pointer at (2i,2j)*
10	LBL 01		41	RTN	
11	CF 01		42	LBL "CMS"	*X holds (j,i)*
12	XEQ "CMR"		43	ST+ X	*2j,2i*
13	J+		44	MSIJA	
14	FS? 09		45	RDN	
15	SF 01		46	MS	*Re(aij)*
16	FS? 10		47	J-	
17	SF 00		48	I-	
18	ASWAP	*FROM,TO*	49	MSC+	
19	RCL Z	*i,j*	50	X<>Y	*Im(aij)*
20	I<>J	*j,i*	51	MSR+	
21	XEQ "CMS"		52	CHS	*-Im(aij)*
22	ASWAP	*TO,FROM*	53	I-	
23	FS? 00		54	MS	
24	*GTO 02*		55	RTN	
25	RCL Z		56	LBL 02	
26	FS? 01		57	ASWAP	*TO,FROM*
27	INT	*next complex row*	58	MAT=	
28	1,001		59	PURFL	*purges TO*
29	FC? 01	*ran out of columns?*	60	MNAME?	*FROM*
30	FRC	*next complex column*	61	END	*142 bytes*
31	+	*update complex pointer*			

This method can be of further use if the Conjugate Transpose needs to be calculated. To that effect all that would be needed are the instructions { X<>Y, CHS, X<>Y } right after step 20 to negate the imagnary parts – with no impact to the CMR and CMS sub-routines.

Example.- Transpose the 3x3 complex matrix shown below and use **CMEDIT** to review the result.



```
    [ 1+i3   2-i5    7-i1 ]
A=  [ 4-i2   6+i9   -8+i4 ]
    [-3-i7   3+i2   -1+i6 ]
```



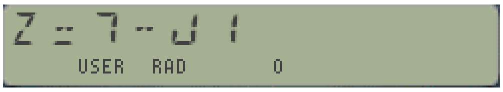
Example of a Complex 3 x 4 Matrix in all its glory:							
Ra11	-Ia11	Ra12	-Ia12	Ra13	-Ia13	Ra14	-Ia14
Ia11	Ra11	Ia12	Ra12	Ia13	Ra13	Ia14	Ra14
Ra21	-Ia21	Ra22	-Ia22	Ra23	-Ia23	Ra24	-Ia24
Ia21	Ra21	Ia22	Ra22	Ia23	Ra23	Ia24	Ra24
Ra31	-Ia31	Ra32	-Ia32	Ra33	-Ia33	Ra34	-Ia34
Ia31	Ra31	Ia32	Ra32	Ia33	Ra33	Ia34	Ra34

Update: Storing and Recalling Complex Matrix Elements. { CMS , CMR }

Revision "5Y+" includes MCODE versions of the **CMS** and **CMR** routines to store and recall complex matrix elements. This may sound like a fairly trivial feat given that all required STO/RCL components are already implemented in MCODE, but because of the way the MCODE were originally written (a direct inheritance from the CCD module) it wasn't possible to chain them without running out of CPU registers, including the RTN address stack.

So a couple of tricks were needed to pull this off, most notably stack register T was needed as scratch – and therefore should not be used when CMS/CMR are used in upir programs or manual calculation. This is not a very limiting restriction (usually T is normally overwritten by natural occurrences, such as ENTERing new values in the stack) but you should bear it in mind for advanced cases.

Note that this limitation does not affect all other Matrix functions, such as MSIJ, MR, MS, J+, etc.
And that in manual more CMR will show the complex value on the display:



Z=7-J1

USER RAD 0

Stack usage for **CMR** and **CMS**

The table below summarizes the Input/Output assumptions for these functions. Note that contrary to their real-matrix counterparts MR and MS, the complex versions require the complex pointer in the X-Register. This pointer will be saved in LastX so you can retrieve it in case it's needed for further operations.

Register	**CMS**		**CMR**	
	Input	Output	Input	Output
X	Cpx. Pointer	Re(z)	Cpx. Pointer	Re(z)
Y	Re(z)	Im(z)	-	Im(z)
Z	Im(z)	0	-	Y
T	-	0	-	Z
L	-	Cpx. Pointer	-	Cpx. Pointer

With this convention you can chain Recall/Storage actions easily, forr instance the sequence of instructions below will copy complex element z(1,2) into z(2,1):

```
1,002, CMR, LASTX, I<>J, CMS
```

Finally, note that

- **CMR** and **CMS** have been promoted to the Main FAT in the upper page of the module. They traded places with two little-used routines (**MRND** and **PSCAL**), which now reside in the auxiliary FAT for subfunctions. Be aware of the XROM numbers displacement in your FOCAL programs

- If there's a data error condition (UNDEFined Array, END of Array, etc.) the register T will contain a non-normalized number of the form: "F|000000ADDR|000". You should disregard this value.

- Upon successful execution, the complex pointer (i,j) will remain selected in the matrix on completion of the action, which is the same as saying that the real pointer (2i,2j) will be active.

  Cpx(i,j) = [(2i-1, 2j-1) ; (2i-1, 2j)
  (2i, 2j-1) ; (2i, 2j)]

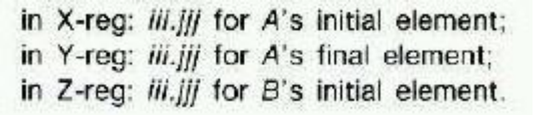
2.3.3.- Other Matrix Functions (“Utilities”)

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.

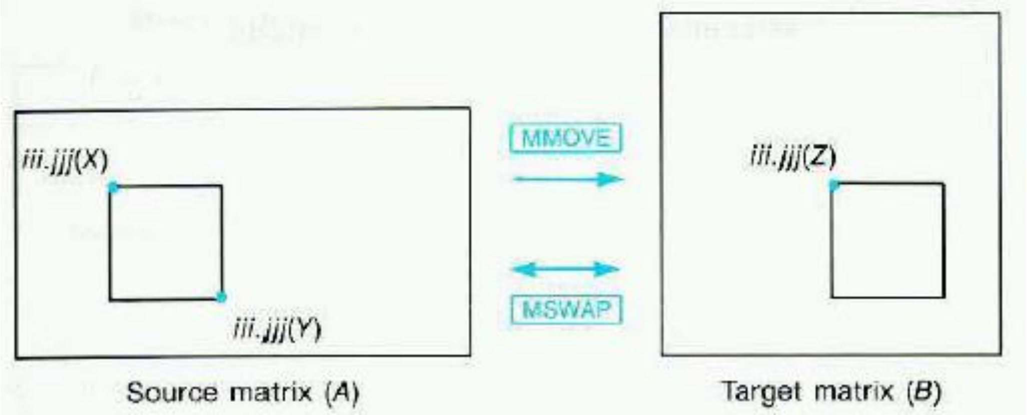
	Function	Description	Input
1	**C<>C**	Exchange columns k and l in a matrix	Name in Alpha, kkk.lll in X-reg
2	**R<>R**	Exchange Rows k and l in a matrix	Name in Alpha, kkk.lll in X-reg
3	**MMOVE**	Matrix Move	Names in Alpha, Pointers in stack
4	**MSWAP**	Matrix Swap	Names in Alpha, Pointers in stack

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.

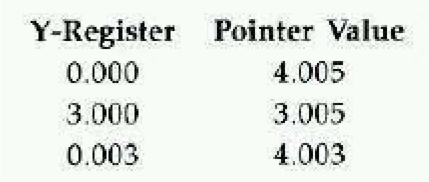
When executing **MMOVE** and **MSWAP** if **A** and **B** are the same matrix and the source submatrix overlaps the target submatrix, the elements are processed in the following order: reverse column order (last to first) and reverse element order (last to first) within each column.



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:

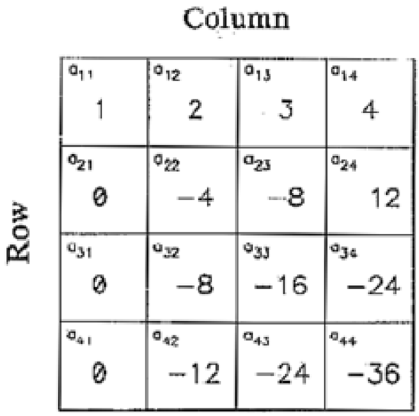


Y-Register	Pointer Value
0.000	4.005
3.000	3.005
0.003	4.003

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

Examples. Calculate the different maxima and minima for the following matrix:



Column

Row

a_{11} 1	a_{12} 2	a_{13} 3	a_{14} 4
a_{21} 0	a_{22} −4	a_{23} −8	a_{24} 12
a_{31} 0	a_{32} −8	a_{33} −16	a_{34} −24
a_{41} 0	a_{42} −12	a_{43} −24	a_{44} −36

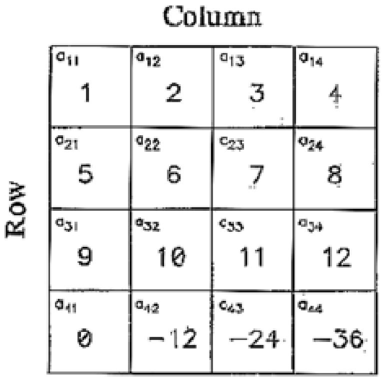
Function	Result Pointer	Result value
MAX	a14	12
MAXAB	a44	36
CMAXAB	a43	24
RMAXAB	a24	12
MIN	a44	-36

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 column of matrix.	Matrix name in Alpha. Outputs column norm to X-reg. Sets pointer to first element of column.
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.

Examples. Calculate the Colum and Rows Sums for the matrix below, using the names "RSUM" and "CSUM" for the result matrices.



Row \ Column				
	a_{11} 1	a_{12} 2	a_{13} 3	a_{14} 4
	a_{21} 5	a_{22} 6	c_{23} 7	a_{24} 8
	a_{31} 9	a_{32} 10	c_{33} 11	a_{34} 12
	a_{41} 0	a_{42} −12	c_{43} −24	a_{44} −36

10
26
42
-72

The result matrices (vectors) are located at the bottom and right of the original matrix.

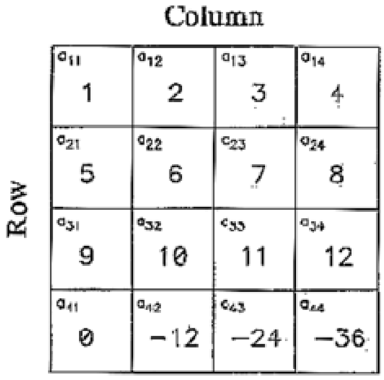
Note that those must have been created previously using MATDIM, or otherwise the "UNDEF ARRAY" will be shown.

15	**6**	**-3**	**-12**

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 column 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.

Examples.- Calculate the pivot element under the second column and compare the first and third rows for the matrix below.



Row \ Column				
	a_{11} 1	a_{12} 2	a_{13} 3	a_{14} 4
	a_{21} 5	a_{22} 6	c_{23} 7	a_{24} 8
	a_{31} 9	a_{32} 10	c_{33} 11	a_{34} 12
	a_{41} 0	a_{42} −12	c_{43} −24	a_{44} −36

Input	Function	Pointer	Output
2	**PIVOT**	a42	12
1,003	**R>R?**		"NO"

Both **PIV** and **YC+C** are vestigial from the CCD "-ARRAY FNS" group, and arguably not very useful now that the major matrix operations are also available.

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 values 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 matrix

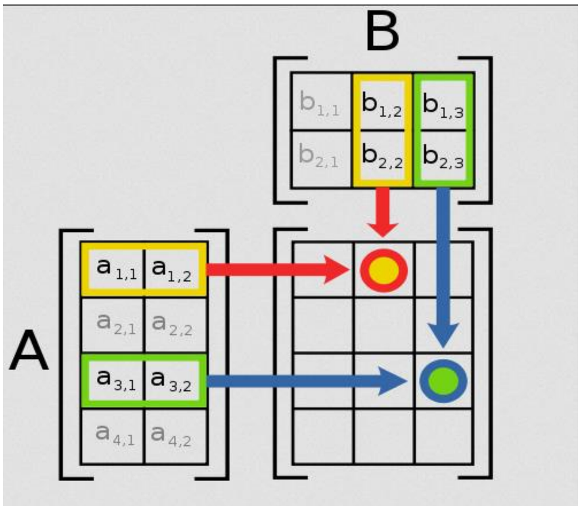
$$A = \begin{bmatrix} 3 & 5 & 7 \\ 2 & 6 & 4 \\ 0 & 2 & 8 \end{bmatrix}, \qquad \|A\|_F = \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n}|a_{ij}|^2}$$

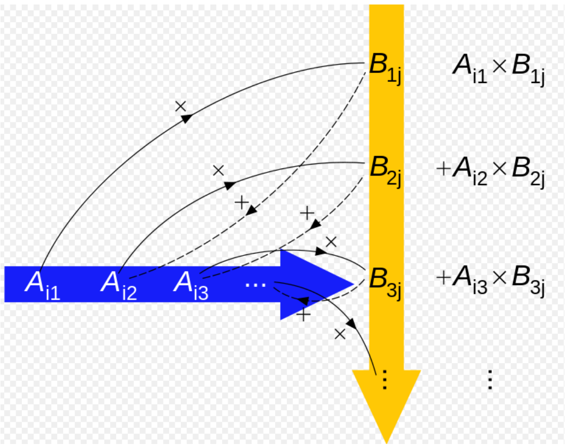
$\|A\|_1 = \max_{1\leq j\leq n}\sum_{i=1}^{m}|a_{ij}|$, which is simply the maximum absolute column sum of the matrix.

$\|A\|_\infty = \max_{1\leq i\leq m}\sum_{j=1}^{n}|a_{ij}|$, 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.





Appendix. Pascal and Random matrices. { PSCAL , MRND }

In mathematics, particularly matrix theory and combinatory, the Pascal matrix is an infinite matrix containing the binomial coefficients as its elements. There are three ways to achieve this: as either an upper-triangular matrix, a lower-triangular matrix, or a symmetric matrix. The 5×5 truncation of these is shown below.

$$S_5 = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 2 & 3 & 4 & 5 \\ 1 & 3 & 6 & 10 & 15 \\ 1 & 4 & 10 & 20 & 35 \\ 1 & 5 & 15 & 35 & 70 \end{pmatrix}$$

The elements of the symmetric Pascal matrix are the binomial coefficients, i.e.

$$S_{ij} = \binom{n}{r} = \frac{n!}{r!(n-r)!}, \quad \text{where } n = i + j, \quad r = i.$$

in other words, see the relationships for the general term and the matrix trace below:

$$S_{ij} = {}_{i+j}\mathrm{C}_i = \frac{(i+j)!}{(i)!(j)!}. \qquad \mathrm{tr}(S_n) = \sum_{i=1}^{n} \frac{[2(i-1)]!}{[(i-1)!]^2} = \sum_{k=0}^{n-1} \frac{(2k)!}{(k!)^2}$$

Random matrices are implemented in the SandMatrix using a time-based seed, applying a pseudo-random algorithm on it and then summing the digits of the mantissa to come up with an integer number. It is furthermore rounded to the current display settings, thus you can limit the values as required. Functions **SEEDT**, **RAND**, and **ΣDGT** in the SandMath, are used.

See the program listings below. For the Pascal matrix, function **NCR** in the SandMath calculates each element value. Note how the subindices are tweaked to return ones in the first row and column – a nice trick courtesy of JM Baillard.

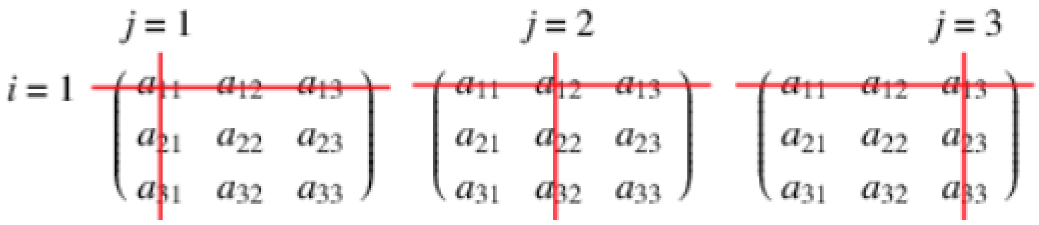
1	LBL "PSCAL"	*MNAME in Alpha*
2	*0*	
3	MSIJA	*set pointer to 1:1*
4	LBL 00	
5	MRIJ	*recall pointer*
6	INT	*i*
7	MRIJ	*recall pointer*
8	FRC	*0,jjj*
9	I<>J	*j*
10	+	*I+j*
11	*2*	
12	-	*i+j-2*
13	MRIJ	*recall pointer*
14	INT	*i*
15	*E*	
16	-	*i-1*
17	NCR	*C(i+j-2; i-1)*
18	MSR+	
19	FC?10	*end of matrix?*
20	*GTO 00*	*no, loop back*
21	END	*yes, done*

1	LBL "MRND"	*MNAME in Alpha*
2	*0*	
3	MSIJA	*set pointer to 1:1*
3	SEEDT	*time-based seed*
4	LBL 00	
6	RAND	*random number*
7	RND	*current settings*
8	Σ DGT	*sum of digits*
9	MSR+	*store element*
10	FC? 10	*end of matrix?*
11	*GTO 00*	*no, loop back*
12	END	*yes, done*

Appendix.- Matrix Minors. { MINOR , CMINOR }

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

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



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

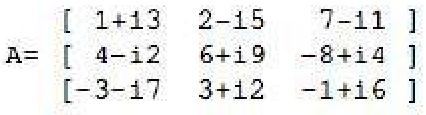
Program listing.- Real Matrix Minors

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

Program listing: Complex Matrix Minors.

1	LBL "CMINOR"		33	ST+ X	*2i,2j*
2	LBL 01		34	STO 02	
3	STO 00		35	XEQ 03	
4	ASTO 01		36	RCL 02	*2i,2j*
5	*"|-,#1"*		37	1,001	
6	**MAT=**	*scratch copy*	38	-	*(2i-1),(2j-1)*
7	**DIM?**		39	LBL 03	
8	2,002		40	RCL 00	*i,j*
9	-	*one order less*	41	FS? 00	
10	*"#2"*		42	**I<>J**	
11	**MATDIM**	*scratch sub-array*	43	INT	
12	**MZERO**	*clear it*	44	DSE X	*discard first column*
13	*"#1"*		45	X=0?	
14	RCL 00		46	RTN	*don’t bother if j=1*
15	CF 00	*do the rows*	47	X<>Y	
16	XEQ 02		48	INT	
17	RCL 00		49	ENTER^	
18	**I<>J**	*i,j pointer*	50	ENTER^	
19	SF 00	*do the columns*	51	2	
20	XEQ 02		52	-	
21	CLST		53	**I<>J**	
22	3,003		54	+	*k,00(k-2)*
23	*"#1,#2"*		55	LBL 00	
24	**MMOVE**		56	FS? 00	
25	PURFL		57	**C<>C**	*bubble left column*
26	CLA		58	FC? 00	
27	**XROM "CMDET"**		59	**R<>R**	*bubble up row*
28	CLA		60	2.002	*offset*
29	ARCL 01		61	-	*k,00(k-2)*
30	RTN		62	DSE Y	*j=j-1*
31	*GTO 01*		63	*GTO 00*	
32	LBL 02		64	**END**	

Example: Calculate all element minors for the example matrix used in the Complex Transposed dexample:



```
    [ 1+i3   2-i5    7-i1  ]
A=  [ 4-i2   6+i9   -8+i4  ]
    [-3-i7   3+i2   -1+i6  ]
```

You need to provide the matrix name in ALPHA and *the complex pointer value in X* - i.e. from 1,001 to 3,003 in this example*.* Also remember that the 41Z Module needs to be plugged for the complex determinat calculation.

The solutions are:

	-28+J31	-44-J18	-29+J71
Minors:	5+J6	9+J49	38+J10
	-47-J9	-46-J2	-19+J51

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

-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, the most remarkable one is of course **PMTM** – which expedites element data entry to the maximum possible on 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 pressing 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 matrix dimension), thus the prompts will continue to appear until the last row is completed. A row counter is added to the prompt to indicate the current row being edited.

If you enter fewer elements in the prompt than existing columns, the remaining 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** the 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 conditions above are respected.

At the heart of **PMTM** there is the function **^MROW** ("Enter Matrix Row"), responsible for the presentation of the prompt in Alpha and accepting the keyboard inputs 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.

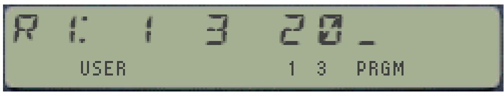
^MROW is called in a loop as many times as rows exist in the matrix, while **ANUMDL** (in the SandMath) is used every iteration (each time a row is being processed) to "extract" the individual element data from the global string in the prompt.

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.

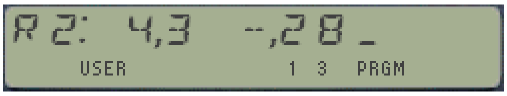
1	**LBL "PMTM"**	
2	**0**	
3	**MSIJA**	*position pointer to 1.1*
4	**LBL 01**	
5	**MRIJ**	*recall pointer*
6	**INT**	*row number*
7	**^MROW**	*prompts for string*
8	**CF 22**	*default reset*
9	**LBL 00**	*separate elements*
10	**ANUMDL**	
11	**FC?C 22**	*last one reached?*
12	**GTO 02**	*yes, exit*
13	**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 available as a sub-function listed within the aux CATalog – 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 peruse 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:


R 1: 1 3 20_
USER 1 3 PRGM

, and


R 2: 4,3 -,28_
USER 1 3 PRGM

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 **OXR** and **OXC**, two sub-functions (i.e. they require **ΣF$** to launch) to only view one specific row or column.

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: when the matrix is not LU decomposed, 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 the actual element input and review.

1	LBL "OMR"		35	FS? 04	*LU decomposed?*
2	*0*	*clears F0-F7*	36	*GTO XX*	*synthetic jump (!)*
3	*GTO 05*		37	INT?	*integer?*
4	LBL "OMC"		38	AINT	*yes, append IP*
5	*2*	*sets F1*	39	FRC?	*fractional?*
6	*GTO 05*		40	ARCL X	*yes, append all*
7	LBL "IMR"		41	FC? 00	*view only?*
8	*E*	*sets F0*	42	AVIEW	*yes, show it*
9	*GTO 05*		43	FC? 00	*view only?*
10	LBL "IMC"		44	*GTO 02*	*yes, skip editing*
11	*3*	*sets F0 & F1*	45	*"|-?"*	*append "?"*
12	LBL 05		46	PROMPT	*show current value*
13	*XEQ 06*	*check status*	47	MS	*store new value*
14	CLX		48	LBL 02	
15	MSIJA	*resets pointer to 1:1*	49	FC? 01	*by column?*
16	*GTO 00*	*go to first element*	50	J+	*yes, next column*
17	LBL "OXC"		51	FS? 01	*by row?*
18	*E1*	*sets F1 & F3*	52	I+	*yes, increase row*
19	*GTO 04*		53	E1	*F10*
20	LBL "OXR"		54	FS? 03	*by row?*
21	*8*	*sets F3*	55	DSE X	*yes, F9*
22	LBL 04		56	FC? IND X	*end of matrix/row?*
23	*XEQ 06*	*check status*	57	*GTO 00*	*no, next element*
24	RDN	*colum/row number*	58	MNAME?	*yes, recall Mname*
25	INT	*just in case*	59	RTN	*done.*
26	E3/E+	*1,00x*	60	LBL 06	
27	FC? 01	*row?*	61	X<>F	*set case lags*
28	I<>J	*yes, transpose*	62	SF 25	*prepare test*
29	MSIJA	*set pointer to row/col*	63	LU?	*is LU decomposed?*
30	LBL 00		64	NOP	*will clear F25 if so*
31	*"a"*	*element symbol*	65	FC?C 25	*was it LU?*
32	MRIJ	*recall index*	66	SF 04	*yes, flag this fact*
33	MP	*prompt index=*	67	END	*and return*
34	MR	*recal value*			

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 to be already dimensioned.

	Function	Description	Input / Output
1	**MAT=**	Creates & Makes matrix B equal to A: B = A	Matrix names in Alpha: "A,B". Matrix A must already exist.
2	**MATP**	Driver for M*M operation Prompts for element values	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 Alpha, 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.
10	**MZDG**	Zeroes (clears) the diagonal elements in matrix (makes aii = 0 , i= 1, 2..m)	Matrix name in Alpha: "A". The diagonal is zeroed.

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 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. **MZERO** is implemente entirely in MCODE.

A related function is **MZDG** , which only deletes the diagonal elements. It can be used to construct an anti-Identity matrix, applying it to a all-ones matrix - created using the three-step sequence: { 1, **MCON**, **MZDG** }. These matrices have the interesting (unproven) property that their determinants obey the expression:

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

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*
2	SIZE?	*current SIZE*
3	MSZE?	*matrix size*
4	E	
5	+	*plus one*
6	X>Y?	*is it larger?*
7	PSIZE	*yes, adjust size*
8	*"|-,R"*	*prepare Alpha string*
9	MAT=	*make matrix R0 equal*
10	MSZE?	*its size again*
11	E3/E+	*prepare control word*
12	RGSORT	*sort registers*
13	ASWAP	*swap alpha*
14	CLST	*prepare pointers*
15	MMOVE	*move all elements*
16	MNAME?	*recall original name*
17	END	*done*

1	LBL "MZERO"	*MNAME in Alpha*
2	DIM?	*get dimension*
3	SF 25	
4	PURFL	*purge file*
5	FC?C 25	*was in main mem*
6	GTO 01	*jump over*
7	MATDIM	*re-create file*
8	RTN	*done*
9	LBL 01	
10	ANUM	*get first reg from title*
11	ENTER^	*copy in Y-reg*
12	MSZE?	*get matrix size*
13	+	*add to first reg*
14	E3/3+	*prepare index format*
15	+	*add to first reg*
16	CLRGX	*clear registers*
17	END	*done*

MSZE? has a new MCODE implementation in this revision – directly reading the matrix header register. Its functionality is equivalent to **FLSIZE** for matrices stored in X-mem – and of course it also does the equivalent for matrices stored in main memory. **MSZE?** is implemented entirely in MCODE.

1	MSZE?	*Header*	A616	*0BF*	**"?"**	
2	MSZE?	*Header*	A617	*005*	**"E"**	***Matrix Size?***
3	MSZE?	*Header*	A618	*01A*	**"Z"**	
4	MSZE?	*Header*	A619	*013*	**"S"**	
5	MSZE?	*Header*	A61A	*00D*	**"M"**	*Ángel Martin*
6	MSZE?	**MSZE?**	**A61B**	*379*	*PORT DEP:*	***Jumps to Bank_2***
7	MSZE?		A61C	*03C*	*XQ*	*adds "4" to [XS]*
8			A61D	*1D9*	*->A5D9*	*[LNCH0]*
9	*valid for main and X-mem*		A61E	***388***	***<parameter>***	*B788*
10	*the proper way to do it!*		A61F	*00B*	*JNC +01*	
11			A620	**100**	**ENROM1**	*restore bank-1*
12	MSZE?		A621	0B0	C=N ALL	*header register*
13	MSZE?		A622	106	A=C S&X	
14	MSZE?		A623	*17D*	*?NC GO*	*[BIN-BCD] plus [RCL]*
15	MSZE?		A624	*0C6*	*->315F*	*[ATOX20]*

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. **MIDN** *is implemented entirely in MCODE*.

The code 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 they don’t require scratch matrices for intermediate calculations. See 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	O		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 functions perhaps **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 one with zeroes when 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.

MATP 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.

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.

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*

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

The next group includes 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.
	MPOL	Matrix Polynomial. P(A) = Σ ak [A]^k	Matrix name in Alpha, control word in X Output is matrix "P" in X-Mem
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.

Some 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 foolproof, and assume the user has a general understanding of the subjects – so they won't be used foolishly.

Matrix Exponential { 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.

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

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:

$$Ln(A) = (A - I) - (A - I)^2/2 + (A - I)^3/3 - (A - I)^4/4 + \ldots\ldots$$ 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.

```
        [[ 1    2    3]
   A =   [ 0    1    2]
         [ 1    3    2]]
```

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

```
          [[19.45828375   63.15030507   66.98787675]
 exp(A) =  [ 8.534640269  32.26024414   33.27906416]
           [ 16.63953207  58.45323648   61.70173665]]
```

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:

```
        [[ 1.2    0.1    0.3]
   A =   [ 0.1    0.8    0.1]
         [ 0.1    0.2    0.9]]
```

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

The result matrix after executing MLN is as follows:

```
          [[ 0.167083396   0.069577923    0.287707999]
 Ln(A) =   [ 0.097783005  -0.240971674    0.103424021]
           [ 0.086500972   0.235053124   -0.131906636]]
```

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

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 could 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 Denman-Beavers iteration. Let Y0 = A and Z0 = I, where I is the n × n identity matrix. The iteration is defined by

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

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:

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

We’ll use IMR to input the element values (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

```
             [[0,917390290  0,345469265  0,197600714]
  Sqrt(A) =   [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:

```
               [[0,999999999  0,499999999  0,333333333]
  [Sqr(A)]^2 =  [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**) registers needed either in main or X-memory (including the original matrix).

1	LBL "MSQRT"		30	X=YR?	*are they equal>*
2	SQR?	*square?*	31	SF 00	*yes, flag this fact*
3	LU?	*yes but LU?*	32	X=YR?	*are they equal>*
4	-ADV MATRX	*not square, show error*	33	*GTO 02*	*yes, jump over*
5	CF 00		34	CLA	*no, keep at it*
6	FNRM	*initial norm*	35	ARCL 01	
7	STO 00	*store it in R00*	36	*"|-,#"*	*prepare Alpha string*
8	ASTO 01	*matrix name to R01*	37	MINV	*invert matrix*
9	DIM?	*dimension to X-reg*	38	MAT=	*copy in auxiliary*
10	*"P"*		39	MINV	*undo the inversion*
11	MATDIM	*auxiliary matrix P*	40	*"Q.#,Q"*	
12	*"Q"*		41	MINV	*invert auxiliary*
13	MATDIM	*auxiliary matrix Q*	42	MAT+	*sum it to partial result*
14	MIDN		43	*"Q,X"*	
15	LBL 00		44	2	
16	*"Q,#"*		45	MAT/	*divide by scalar 2*
17	MINV		46	LBL 02	
18	MAT=	*auxiliary matrix #*	47	*"P,"*	
19	CLA		48	ARCL 01	
20	ARCL 01		49	MAT=	
21	*"|-,#,P"*		50	FC? 00	*were norms equal?*
22	MAT+		51	*GTO 00*	*no, next iteration*
23	*"P,X"*		52	PURFL	*purge P*
24	2		53	*"Q"*	
25	MAT/		54	PURFL	*purge Q*
26	FNRM	*Frobenius norm*	55	*"#"*	
27	VIEW X	*show progress*	56	PURFL	*purge #*
28	X<> 00	*swao with old norm*	57	MNAME?	*matrix name to Alpha*
29	RCL 00	*recall new one again*	58	END	*done*

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

Appendix.- Square root of a 2x2 Matrix. { MSQ2 }

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:

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

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.-

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

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.

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

or numerically: **sqr(A) =** [[2.944271912 -0.472135955]
[1.416407865 1.291796068]]

Matrix Integer Powers and Roots. { M^2 , MPWR , 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.

MPWR calculates the general case n, whilst 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.

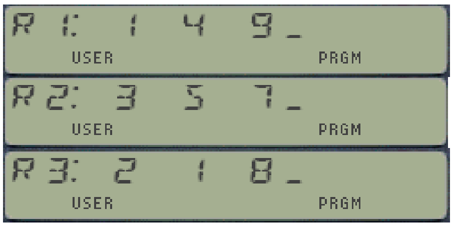
The exponent may also be a negative integer. For that case the inverse matrix is calculated first, and the positive integer power is used for it. Lastly, for n=0 the result is the identity matrix of course.

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

$[A]\wedge^{1/x} = \exp[1/x\ .\ \mathrm{Ln}(A)]$, 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:



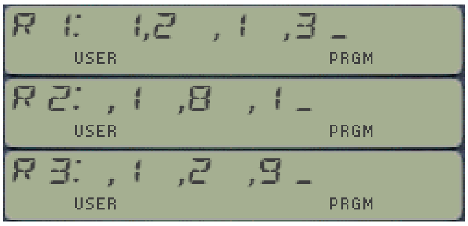
```
R 1:  1   4   9_
   USER          PRGM
R 2:  3   5   7_
   USER          PRGM
R 3:  2   1   8_
   USER          PRGM
```

```
        [[ 1     4     9 ]
A =      [ 3     5     7 ]
         [ 2     1     8 ]]
```

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

```
         [[ 7851276     8652584     31076204 ]
A^7 =     [ 8911228     9823060     35267932 ]
          [ 5829472     6422156     23076808 ]]
```

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



```
R 1:  1,2  ,1  ,3_
   USER          PRGM
R 2:  ,1  ,8  ,1_
   USER          PRGM
R 3:  ,1  ,2  ,9_
   USER          PRGM
```

```
        [[ 1.2   0.1    0.3]
A =      [ 0.1   0.8    0.1]
         [ 0.1   0.2    0.9]]
```

The results are as follows:

```
           [[1,034632528   0,015156701   0,057916477]
A^1/5 =     [0,019601835   0,953558110   0,020490861]
            [0,017823781   0,045426856   0,974937998]]

           [[1,199999994   0,100000000   0,300000000]
[A^1/5]^5 = [0,100000000   0,800000000   0,100000000]
            [0,100000000   0,200000000   0,900000000]]
```

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

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

1	LBL "M^2"	*MNAME in Alpha*
2	SQR?	*square?*
3	LU?	*yes but LU?*
4	-ADV MATRX	*not square, show error*
5	-CCD MATRX	*no, show 'Running…"*
6	ASTO L	
7	DIM?	*dimension to X-reg*
8	*"|-,"*	
9	ARCL L	
10	*"|-,P"*	*"M,M,P"*
11	ASWAP	*"M,P,M""*
12	ASWAP	*"P,M,M"*
13	MATDIM	*auxiliary P*
14	ASWAP	*"M,M,P"*
15	M*M	*matrix product*
16	CLAC	*"M,M,"*
17	CLAC	*"M,"*
18	*"|-P"*	*"M,P"*
19	ASWAP	*"P,M"*
20	MAT=	*result to M*
21	PURFL	*purge P*
22	MNAME?	*MNAME to Alpha*
23	END	*done*

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

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	SQR?	*square?*	22	"#"	*no*
5	LU?	*yes but LU?*	23	PURFL	*get rid of scratch*
6	-ADV MATRX	*not square, 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	DIM?	*dimansion to X-reg*	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 (and you have nothing else to do) 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.

DIVRGNT

A general-purpose algorithm for the p-th. root { M^1/N }

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

M0 = A ; Mk+1 = Mk .{ (2.I + (p-2) Mk) (I + (p-1) Mk)^(-1) }^p

X0 = I ; Xk+1 = Xk (2.I + (p-2) Mk)^(-1) . (I + (p-1) Mk)

where I is the Identity matrix

Mk tends to I as k tends to infinity
Xk tends to A^1/p as k tends to infinity

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

Implementing this algorithm requires *six* scratch matrices, mostly due to the fact that the matrix product **M*M** requires the result matrix to be different from both factors. The total byte count is low but the memory requirement is a hefty toll because of this.

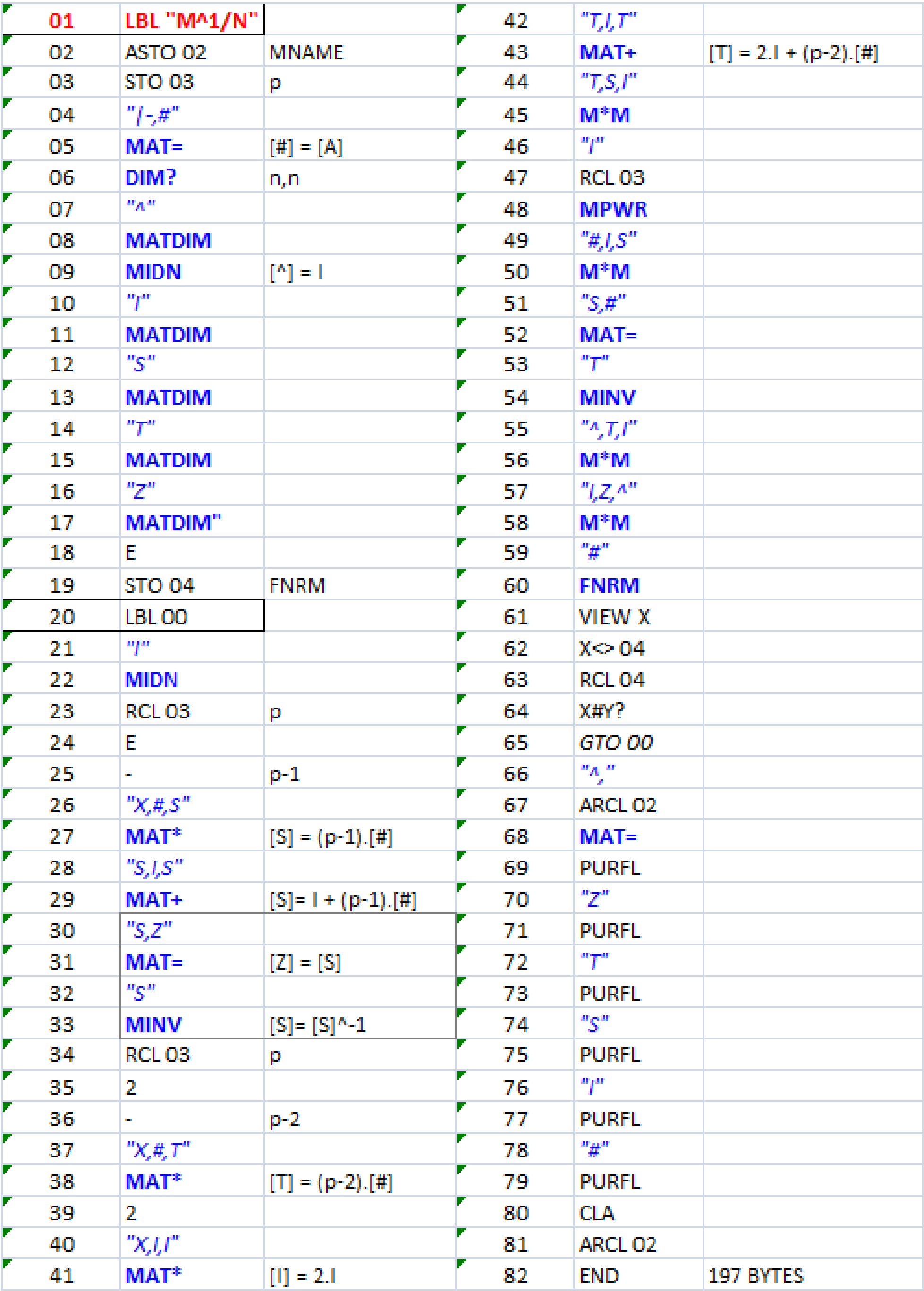
[Z] = I + (p-1).Mk, and: [Y] = 2.I + (p-2).Mk

and using these the algorithm becomes:

Mk+1 = Mk. { [Y] . ([Z]^(-1) }^p
Xk+1 = Xk . Y^(-1) . [Z]

The listing below shows the program included in the module. Note that in addition to those six scratch matrices, the function **MPWR** uses another two, which brings the total count to *eight scratch matrices*. This makes this approach only valid for small order matrices, as the total memory requirement will be 8 times the size of the original matrix.

If the original matrix is in the data registers area, then the maximum order is 8x8 with the 8 auxiliary matrices stored in the X-Mem section. - Admittedly a little crazy, but it’s better than nothing.



01	LBL "M^1/N"		42	"T,I,T"	
02	ASTO 02	MNAME	43	MAT+	[T] = 2.I + (p-2).[#]
03	STO 03	p	44	"T,S,I"	
04	"|-,#"		45	M*M	
05	MAT=	[#] = [A]	46	"I"	
06	DIM?	n,n	47	RCL 03	
07	"^"		48	MPWR	
08	MATDIM		49	"#,I,S"	
09	MIDN	[^] = I	50	M*M	
10	"I"		51	"S,#"	
11	MATDIM		52	MAT=	
12	"S"		53	"T"	
13	MATDIM		54	MINV	
14	"T"		55	"^,T,I"	
15	MATDIM		56	M*M	
16	"Z"		57	"I,Z,^"	
17	MATDIM"		58	M*M	
18	E		59	"#"	
19	STO 04	FNRM	60	FNRM	
20	LBL 00		61	VIEW X	
21	"I"		62	X<> 04	
22	MIDN		63	RCL 04	
23	RCL 03	p	64	X#Y?	
24	E		65	GTO 00	
25	-	p-1	66	"^,"	
26	"X,#,S"		67	ARCL 02	
27	MAT*	[S] = (p-1).[#]	68	MAT=	
28	"S,I,S"		69	PURFL	
29	MAT+	[S]= I + (p-1).[#]	70	"Z"	
30	"S,Z"		71	PURFL	
31	MAT=	[Z] = [S]	72	"T"	
32	"S"		73	PURFL	
33	MINV	[S]= [S]^-1	74	"S"	
34	RCL 03	p	75	PURFL	
35	2		76	"I"	
36	-	p-2	77	PURFL	
37	"X,#,T"		78	"#"	
38	MAT*	[T] = (p-2).[#]	79	PURFL	
39	2		80	CLA	
40	"X,I,I"		81	ARCL 02	
41	MAT*	[I] = 2.I	82	END	197 BYTES

Note: The character “#” represents the standard HP-41 symbol denoting “not equal to”, ASCII value 29. It’s not the same one used in MPWR, which is the ASCII value 35, the “hash” or pound sign.

Lie Product of two Matrices. { MLIE }

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

$$\mathrm{Lie}(A,B) = -\,\mathrm{Lie}(B,A) = AB - BA$$

Example.- Calculate the Lie product for matrices:

```
A =     [[ 1   2   4 ]          and:        B=     [[ 1   4   1 ]
         [ 3   5   7 ]                              [ 5   9   2 ]
         [ 7   9   8 ]]                             [ 6   5   3 ]
```

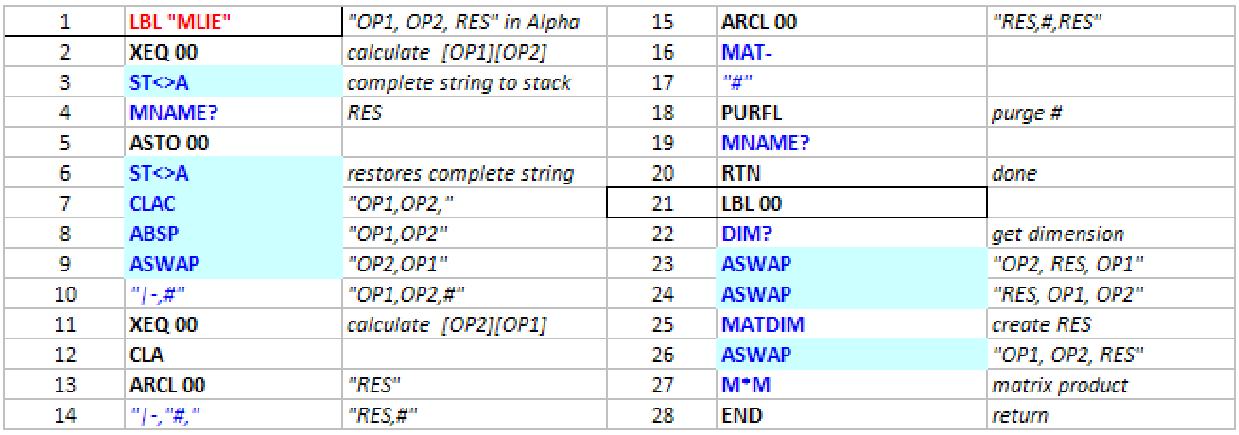
The results are:

```
ALPHA, “A,B,C”, ALPHA                          [[ 15   11   -23 ]
XEQ “MLIE”            ->  Lie(A,B) =            [ 24   19   -65 ]
                                                [ 58   85   -34 ]]

ALPHA. “B,A,C”, ALPHA                          [[-15  -11    23 ]
XEQ “MLIE”            ->  Lie (B,A) =          [-24  -19    65 ]
                                               [-58  -85    34 ]]
```

The program listing is shown on the left. Note the usage of the scratch matrix “#” to temporarily 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** exchanges the contents of the Alpha register around the ***first*** *comma* character encountered; which makes it so interesting in this case.



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

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.,

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

where aii represents the entry on the i-th row and i-th 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:

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

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

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

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

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

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 (courtesy of Thomas Klemm):

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


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

Note that as of revision “Y” **MTRACE** is an MCODE function. Here’s the sweet and short equivalent FOCAL program listing, compared side-by-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 in the alternative approach function **SUM** is used, which removes the need to calculate the last step of the sketch. Regardless, it’s bigger and takes longer execution time, even without the test for square matrix condition.

1	LBL "MTRACE"	*MNAME in Alpha*	1	LBL "TRACE2"	
2	SQR?	*square?*	2	"|-,#"	*prepare Alpha string*
3	LU?	*yes but LU?*	3	MAT=	*make scratch*
4	-ADV MATRX	*not square, 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 }

This function is used to modify the values of all elements, dividing each row by its diagonal element; that is: $a_{ij} = a_{ij} / a_{ii}$, $j=1,2,\ldots n$

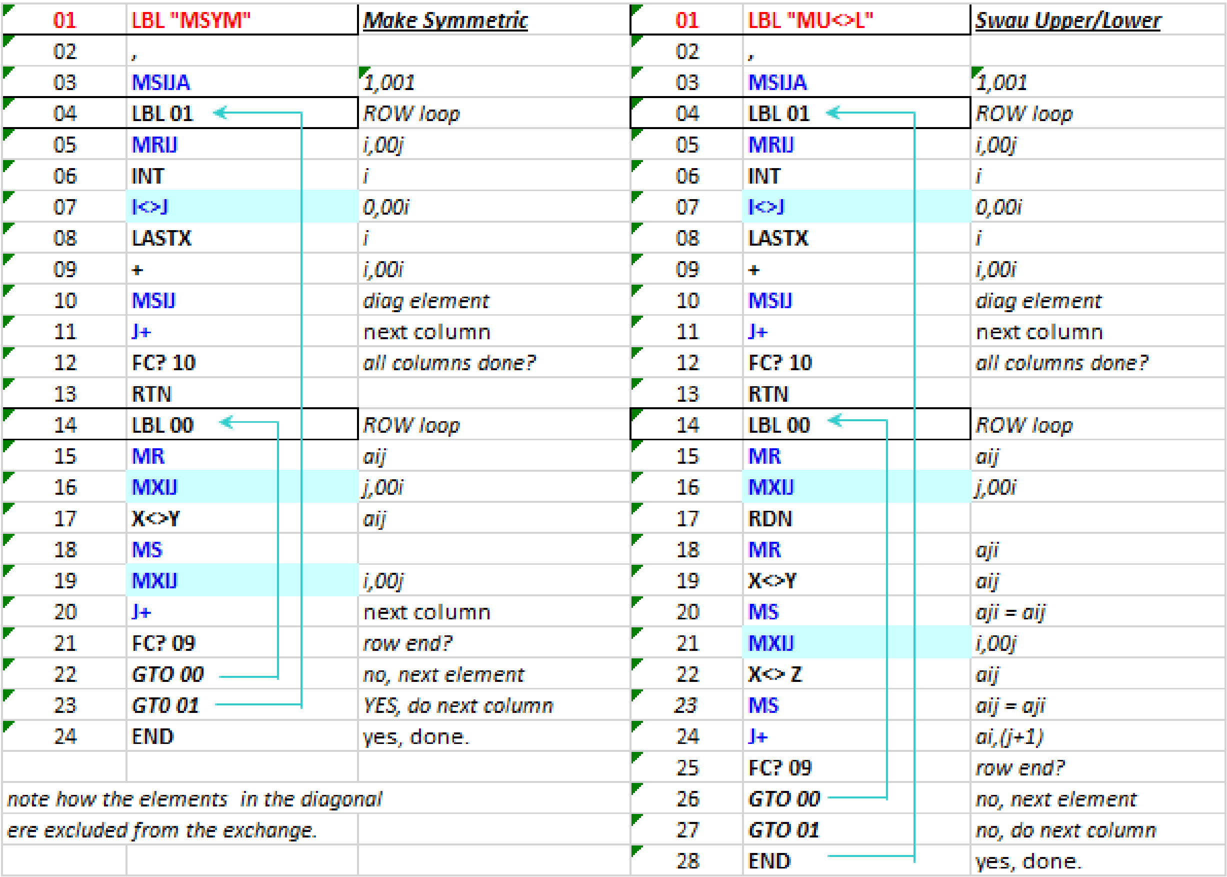
In effect the result matrix has all its diagonal elements equal to 1 (i.e. diagonal is unitary). This type of calculation is useful for row simplification steps in matrix reductions; more like a vestigial function from when the major matrix operations were not available (i.e. the CCD days, pre-Advantage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

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

Transposing elements with { MXIJ }

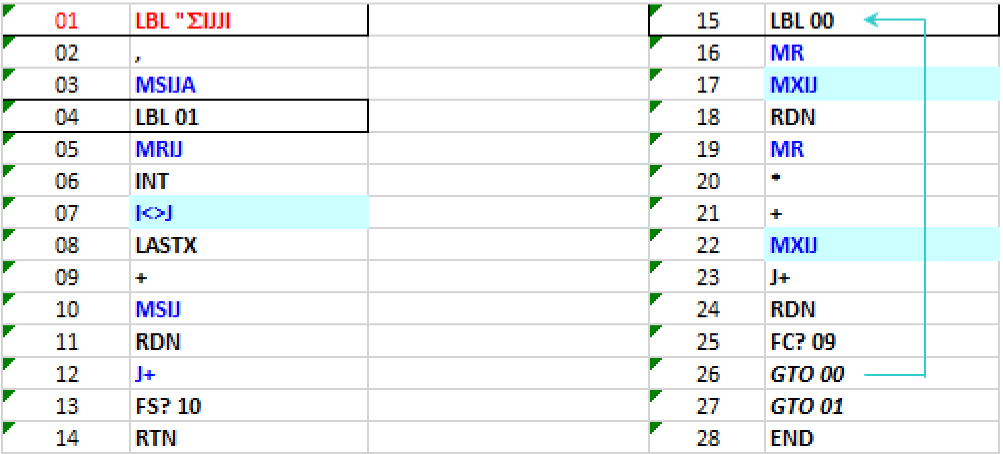
A new function is included to facilitate the elmenet transposition. **MXIJ** exchanges the row and column of the selected element, returning the new selected element pointer to the X-Register. The matrix can be non-square, but an error message will show if the “transposed” pointer does not exist. Note that there’s no need to recall the current pointer first.

The function does the equivalent to the following FOCL snippet: { MRIJ, I<>J, MSIJ }, which is simple enough but having it as a single function allows simplified FOCAL programs. See for example the routines below to make a matrix symmetric (using the upper triangular part), and to swap the upper and lower triangular parts of a square matrix.



01	LBL "MSYM"	Make Symmetric	01	LBL "MU<>L"	Swau Upper/Lower
02	,		02	,	
03	MSIJA	1,001	03	MSIJA	1,001
04	LBL 01	ROW loop	04	LBL 01	ROW loop
05	MRIJ	i,00j	05	MRIJ	i,00j
06	INT	i	06	INT	i
07	I<>J	0,00i	07	I<>J	0,00i
08	LASTX	i	08	LASTX	i
09	+	i,00i	09	+	i,00i
10	MSIJ	diag element	10	MSIJ	diag element
11	J+	next column	11	J+	next column
12	FC? 10	all columns done?	12	FC? 10	all columns done?
13	RTN		13	RTN	
14	LBL 00	ROW loop	14	LBL 00	ROW loop
15	MR	aij	15	MR	aij
16	MXIJ	j,00i	16	MXIJ	j,00i
17	X<>Y	aij	17	RDN	
18	MS		18	MR	aji
19	MXIJ	i,00j	19	X<>Y	aij
20	J+	next column	20	MS	aji = aij
21	FC? 09	row end?	21	MXIJ	i,00j
22	GTO 00	no, next element	22	X<> Z	aij
23	GT0 01	YES, do next column	23	MS	aij = aji
24	END	yes, done.	24	J+	ai,(j+1)
			25	FC? 09	row end?
note how the elements in the diagonal			26	GTO 00	no, next element
ere excluded from the exchange.			27	GTO 01	no, do next column
			28	END	yes, done.

MXIJ is now used in a re-written version of **ΣIJJI**, featuring shorter code and faster execution:



01	LBL "ΣIJJI		15	LBL 00
02	,		16	MR
03	MSIJA		17	MXIJ
04	LBL 01		18	RDN
05	MRIJ		19	MR
06	INT		20	*
07	I<>J		21	+
08	LASTX		22	MXIJ
09	+		23	J+
10	MSIJ		24	RDN
11	RDN		25	FC? 09
12	J+		26	GTO 00
13	FS? 10		27	GTO 01
14	RTN		28	END

Appendix: Moore-Penrose Matrix Pseudo-Inverse { $\boxed{\text{MPINV}}$ }

In mathematics, and in particular linear algebra, a pseudoinverse A+ of a matrix A is a generalization of the inverse matrix. The most widely known type of matrix pseudoinverse is the Moore–Penrose pseudoinverse, which was independently described by E. H. Moore in 1920, Arne Bjerhammar in 1951 and Roger Penrose in 1955.

For A in {M}(m,n;K) , a pseudoinverse of A is defined as a matrix A^+ in {M}(n,m;K) satisfying all of the following four criteria:

$A A^+A = A$ (AA^+ need not be the general identity matrix,);
$A^+A A^+ = A^+$ (A^+ is a weak inverse for the multiplicative semigroup);
$(AA+)^* = AA^+$ (AA^+ is Hermitian); and
$(A^+A)^* = A^+A$ (A^+A is also Hermitian).

The pseudoinverse exists and is unique: for any matrix A, there is precisely one matrix A^+ that satisfies the four properties of the definition. A matrix satisfying the first condition of the definition is known as a generalized inverse. If the matrix also satisfies the second definition, it is called a generalized reflexive inverse. Generalized inverses always exist but are not in general unique. Uniqueness is a consequence of the last two conditions.

The SandMatrix uses the iterative method known as Ben-Israel & Cohen, defined by the following recurrence expression:

$$A_{i+1} = 2A_i - A_i A A_i,$$

This recursion produces a sequence converging quadratically to the pseudoinverse of A if it is started with an appropriate A0 satisfying [A0][A] = (A0 A)*. The choice A0 = α A* (where 0 < α < 2/σ^2(A), with σ(A) denoting the largest singular value of A) has been argued not to be competitive to other methods (like Greville’s), because even for moderately ill-conditioned matrices it takes a long time before Ai enters the region of quadratic convergence. However, if started with A0 already close to the Moore–Penrose pseudoinverse and A0 A= (A0 A)*, for example A0:= $[(A^*A+ \delta I)^{-1}] A^*$, convergence is fast (quadratic).

The SandMatrix uses the trace of the matrix product $A.A^T$ instead – which has shown stable convergence, even if finicky. Like always, the use of the 41CL in turbo mode or a good PC emulator is most recommended.

$A_0 = \mu A^T$ where A^T = transposed of A, and $0 < \mu <= 2 / \mathrm{Trace}(A.A^T)$

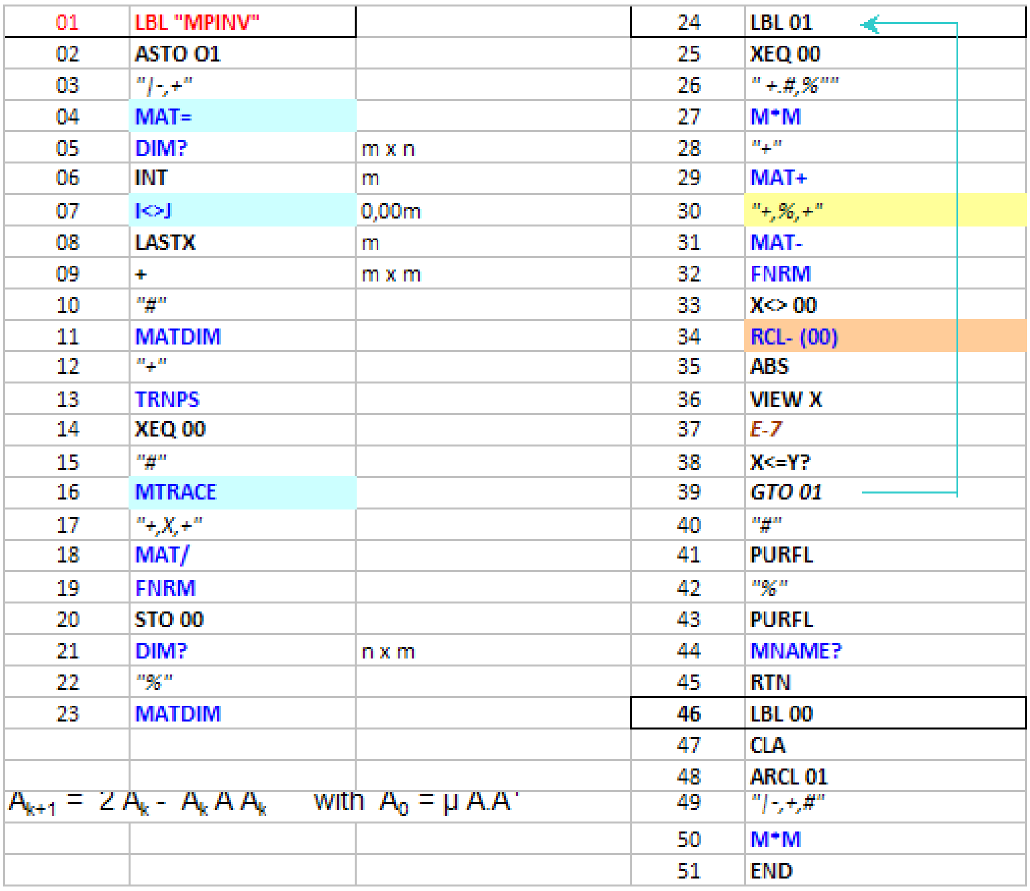
Example: verify that the pseudo inverse of the 3x4 matrix:

$$A = \begin{matrix} [[\,1 & 1 & 4 & 2] \\ [\,0 & 1 & 2 & 3] \\ [\,3 & 2 & 6 & 7]] \end{matrix} \quad , \quad \text{is the 4x3 matrix : } A^+ = (1/112). \begin{matrix} [[\,-21 & -85 & 43\,] \\ [\,7 & 23 & -9\,] \\ [\,49 & 1 & -15\,] \\ [\,-35 & 29 & 13\,]] \end{matrix}$$

The FOCAL program is listed below for your reference – you’ll be surprised to know that after some lengthy conversion of FOCAL code into MCODE the necessary room was freed up in the main bank of module to include it. With just 55 program steps the code is relatively short but it requires two auxiliary matrices for the calculations, besides the result matrix.

As always, the most limiting step is the **M*M** instruction that requires the result matrix to be different from both operands. Note how all the control strings in Alpha drive the operations, and that the default conventions are mostly followed – with a couple of exceptions, like program line #30.

The auxiliary matrices [P] and [Q] are purged upon completion, but the result matrix [M+] does not overwrite the initial one, so both will remain in memory.



01	LBL "MPINV"		24	LBL 01
02	ASTO 01		25	XEQ 00
03	"|-,+"		26	" +.#,%""
04	MAT=		27	M*M
05	DIM?	m x n	28	"+"
06	INT	m	29	MAT+
07	I<>J	0,00m	30	"+,%,+"
08	LASTX	m	31	MAT-
09	+	m x m	32	FNRM
10	"#"		33	X<> 00
11	MATDIM		34	RCL- (00)
12	"+"		35	ABS
13	TRNPS		36	VIEW X
14	XEQ 00		37	*E-7*
15	"#"		38	X<=Y?
16	MTRACE		39	*GTO 01*
17	"+,X,+"		40	"#"
18	MAT/		41	PURFL
19	FNRM		42	"%"
20	STO 00		43	PURFL
21	DIM?	n x m	44	MNAME?
22	"%"		45	RTN
23	MATDIM		46	LBL 00
			47	CLA
			48	ARCL 01
$A_{k+1} = 2 A_k - A_k A A_k$		with $A_0 = \mu A.A'$	49	"|-,+,#"
			50	M*M
			51	END

The instruction RCL- (00) is from the SandMath module, and can be replaced by the pair: { RCL 00, - }

The convergence factor (in line 37) is chosen to be 1e-7 – feel free to change it to a more stringent or a looser one depending on the matrix. The deltas between the matrix norms of the successive iterations will be displayed until the final one is obtained.

The Pseudo-Inverse matrix name is left in Alpha, ready for **OMR, OMC,** or your favorite matrix editor program. The final matrix name is formed by adding the plus sign “+” to the original matrix name.

Warning: The program will leave the pseudo-inverse matrix in X-Memory, regardless of where the original matrix is stored. The reason being the naming convention, which makes the pseude-inverse name not valid for the Standard or CL-registers areas.

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

Other two functions directly related to the eigenvalues are MDPS and ΣIJJI. They compute sums of element pair multiplication, either for those in the diagonal (aii * akk); or for "crossed" (i.e. opposite) ones, (aij * aji), with i#j – excluding the diagonal.

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

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:

$P(x) = 2\ x^4 - x^3 + 3\ x^2 - 4\ x + 5$; 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 *should not 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.

Warning: if you use data registers to store the matrix (with “Rxxx”” name in ALPHA) make sure that the register range does not overlap with the polynomial coefficients.

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"	*Mname in Alpha*	23	*"P,"*	
02	**SQR?**	*is it square?*	24	**ARCL 01**	
03	**LU?**	*LU form?*	25	*"|-,#"*	*"P,A,#"*
04	**-ADV MTRX**	*no, prompt error*	26	**M*M**	
05	**-CCD MTRX**	*shows 'RUNNING…"*	27	*"#,P"*	
06	**E-3**	*cnt'l word in X*	28	**CLST**	
07	**-**		29	**MMOVE**	
08	**STO 00**		30	**ISG 00**	*next index*
09	**ASTO 01**	*saves Mname in R01*	31	***GTO 00***	*loop back*
10	**DIM?**	*dimensions*	32	**XEQ 02**	
11	*"P"*		33	**PURFL**	*purge auxiliary mat*
12	**MATDIM**	*creates scratch*	34	**MNAME?**	*bring result name*
13	*"#"*		35	**RTN**	
14	**MATDIM**	*creates scratch*	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. Not ideal, but at least we double-use it for other intermediate calculations as well (identity matrix products), killing two birds with the same stone.

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

-ADV POLYN
USER 01

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	**EV2X2**	Eigen values 2x2	Subroutine mode
4	**EV3X3**	Eigen values 3x3	Prompts Matrix Elements
5	**JACOBI**	Symmetrical Eigenvalues	Under prgm control

4.1. Eigenvectors and Eigenvaules.

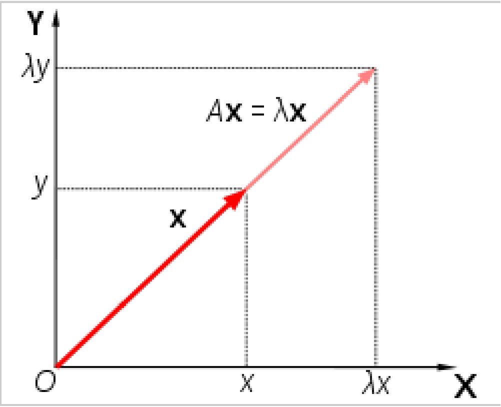
An eigenvector of a square matrix **A** is a non-zero vector ***v*** that, when the matrix is multiplied by ***v***, yields a constant multiple of v, the multiplier being commonly denoted by λ. That is:

$$Av = \lambda v$$

The number λ is 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,

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

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 | λ | > 1, remain the same if | λ | = 1, and decrease it if | λ |< 1. Moreover, the direction will be precisely the same if λ > 0, and flipped if λ < 0. If λ = 0, then the length of the arrow becomes zero.

4.1.1. 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**.

4.1.2. Additional properties of eigenvalues

The trace of A, tr{A}, defined as the sum of its diagonal elements, is also the sum of all eigenvalues, and the determinant of A, det{A} is the product of all its eigenvalues:

$$\operatorname{tr}(A) = \sum_{i=1}^{n} a_{ii} = \sum_{i=1}^{n} \lambda_i \quad ; \quad \det(A) = \prod_{i=1}^{n} \lambda_i$$

These properties are all that’s needed to calculate the egenvalues of a 2x2 matrix, as the solutions of the quadratic eqution: λ^2 – λ.tr(A) + det(A) = 0. See the routine below for details:

```
01  LBL "EV2X2"        ; Expects the Matrix name in ALPHA
02  E                  ; second degree coefficient
03  MTRACE             ; matrix trace
04  CHS                ; first degree coefficient
05  MDET               ; this will leave the matrix in its LU form
06  QROOT              ; quadratic equation roos (in the SandMath)
07  END                ; done – it doesn’t get any simpler than that!
```

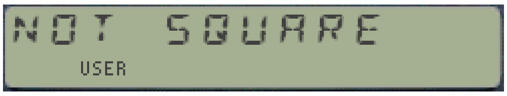
4.1.3 SOLVE-based Implementation. { EIGEN }

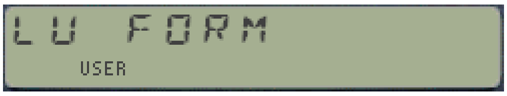
In addition to **EV2X2** there are other three programs in the SandMatrix that calculate eigenvalues. The first one is aptly named **EIGEN**, and is a brute-force approach using the direct definition of the eigenvalue given above. What makes it interesting is the direct application of SOLVE (of **FROOT** 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.

The function is "aware" of whether it's being run in RUN or PROGRAM mode, so that the data entry is omitted during a program execution. There's no need to remember to set user flags or clear it for manual mode, saving one FAT entry for a subroutine as well.

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.


NOT SQUARE
USER


LU FORM
USER

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**"	3.0003	Creates matrix in X-Mem
XEQ "**PMTM**"	"R 1: _"	Prompts for the first row
3, ENTER^, 1, ENTER^, 5, R/S	"R 2: _"	… second row
3, ENTER^, 3, ENTER^, 1, R/S	"R 3: _"	… third row
4, ENTER^, 6, ENTER^, 4, R/S	6.0000	
XEQ "**EIGEN**"	"ORDER=?"	press R/S to use the Matrix in ALPHA
R/S	"LO'V.=?"	Prompts for lower bound
5, R/S	"HI'V.=?"	Higher bound
15, R/S	flying goose…	FROOT is working on it
	"EV= 10,000000"	ev found (in FIX 5).

Note that if you entered a value for the *"ORDER?"* question the program proceeds to create a new matrix and prompt for all its elements. If you want to use the matrix in ALPHA just press "R/S"

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

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.

4.1.4. 3-Dimensional case. { EV3X3 }

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 characteristic 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:

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

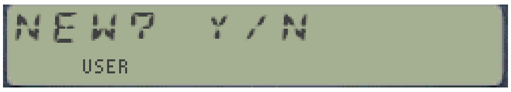
For a 3×3 matrix, the formula specifies the characteristic polynomial to be

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

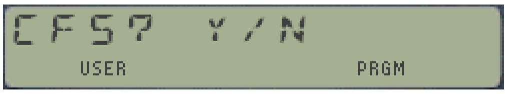
where **c2** is the sum of the principal minors of the matrix = $\frac{1}{2}\left((\operatorname{tr}A)^2 - \operatorname{tr}(A^2)\right)$

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), two choices are offered: 1) to use the matrix in ALPHA or create a new one, and 2) to see the coefficients of the polynomial before calculating its roots (i.e. the eigenvalues). Note that these will only take [Y] / [N] as valid inputs.



NEW? Y/N
USER



CFS? Y/N
USER PRGM

,

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

Solution: pA(x) = 3.25E-8 x3 - 60 x2 -66 x + 1 =0

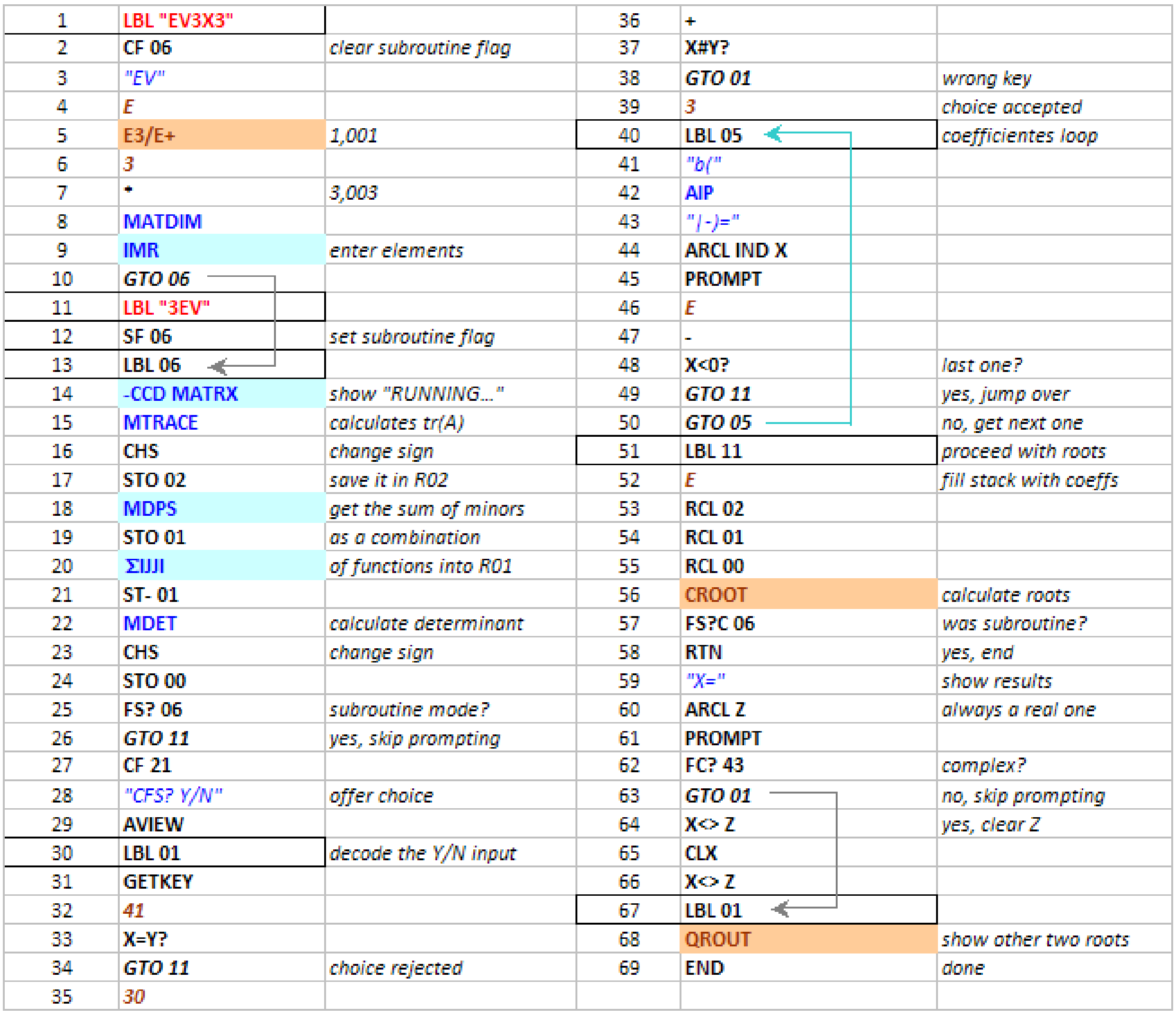
x1 = 66,897
x2 = -0,897
x3 = 2,24000E-9

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

Solution: pA(x) = 2.7426E-9 x3 -18 x2 -15 x = 0

x1= 16,1168
x2 = -1,1168
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.-

- Like EIGEN before, this function behaves differently in RUN and Program modes – skipping the data entry part during a program execution. Note that in manual mode EV3X3 will create a matrix named “EV” if you answer “Y” to the “NEW? Y/N” question, but that in program mode it will work with *any* 3x3 matrix which name is in Alpha. This is compatible with EIGEN in its subroutine mode as well.

- It’s also important to remark that *the used matrix cannot be in LU form* – otherwise the **MDET** function will return an error.

- The three 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.

4.1.5 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. 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 employs the matrix trace in the process. The coefficients are calculated using the iterations:

$b1 = -\mathrm{tr}\,(\mathbf{B}1)$, with $\mathbf{B}1$ = the original matrix, and

$bk = -\,\mathrm{tr}\,(\mathbf{B}k)\,/\,k$, with $\mathbf{B}k = \mathbf{A}(\,\mathbf{B}k\text{-}1 + bk\text{-}1\,\mathbf{I}\,)$, $k=2,\ldots\, n$

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:

$$\mathbf{A} = \begin{bmatrix} 1 & -0.69 & 0.28 \\ -0.69 & 1 & 0.18 \\ 0.28 & 0.18 & 1 \end{bmatrix}$$

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**”	“a 1,1= ?”	Prompts for data, also
1, R/S	“a 1,2= ?”	showing current values
0.69, CHS, R/S	“a 1,3= ?”	
0.28, R/S	“a 2,1= ?”	
0.69, CHS, R/S	“a 2,2= ?”	
1, R/S	“a 2,3= ?”	
0.18, R/S	“a 3,1= ?”	
0.28, R/S	“a 3,2= ?”	
0.18, R/S	“a 3,3= ?”	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”	(round) Coefficient of x^3
if you want AVIEW	“a2= -3”	(round) Coefficient of x^2
to stop each time	“a 1=2.413 100”	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 –

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 results in a much faster execution than the general case using **RTSN** .

1	LBL "CHRPOL"	*MNAME in Alpha*	55	STO 01	*it's monic (!)*
2	SQR?	*is it square?*	55	E3/E+	*1.001*
3	LU?	*yes, but LU form?*	56	+	*1.00(n+1) - cnt'l word*
4	-ADV MATRX	*not square, show error*	57	"#"	
5	ASTO 01	*MNAME*	58	PURFL	
6	-CCD MATRIX	*shows 'RUNNING…"*	59	"P"	
7	"|-,P"		60	PURFL	
8	MAT=	*B = A*	61	PVIEW	*for information*
9	ASWAP		62	-CCD MATRIX	*shows 'RUNNING…"*
10	DIM?	*n,00n*	63	PDEG	*new destination*
11	INT	*n*	64	STO 00	*as expected by RTSN*
12	STO 00		65	4	
13	E		66	X>=Y?	n<=4?
14	+	*n+1*	67	GTO 04	*yes, particular case*
15	MDET	*determinant*	68	CLX	*no, general case*
16	RCL 00		69	E	
17	CHSYX	*independent term*	70	+	*n+1*
18	STO IND Y	*stored in Rn+1*	71	E6	
19	ASWAP		72	/	*0,000|00(n+1)*
20	MAT=	*avoids LU issues*	73	3	*build the "from,to"*
21	DIM?		74	E3/E+	*1.003*
22	"#"	*auxiliary array*	75	+	*1.003|00(n+1)*
23	MATDIM		76	REGMOVE	*as expected by RTSN*
24	FRC	*0,00n*	77	RTSN	
25	2		78	GTO 00	*go to EXIT*
26	+	*2,00n*	79	LBL 04	
27	STO 00		80	X#Y?	*n#4?*
28	CF 21	*not halting VIEW*	81	GTO 03	
29	LBL 00		82	RCL 02	*a3*
30	VIEW 00	*shows index*	83	RCL 03	*a2*
31	"#"		84	RCL 04	*a1*
32	MIDN	*[#] = [I]*	85	RCL 05	*a0*
33	"P"		86	QUART	
34	MTRACE	*tr (B)*	87	GTO 00	*go to EXIT*
35	RCL 00		88	LBL 03	
36	INT	*k+1*	89	RCL 01	*a3*
37	E		90	RCL 02	*a2*
38	-	*k*	91	RCL 03	*a1*
39	/		92	RCL 04	*a0*
40	CHS		93	CROOT	
41	STO IND 00	*pk = -tr (B) / k*	94	"X="	
42	"X,#,#"		95	ARCL Z	
43	MAT*	*[#] = pk [I]*	96	PROMPT	*real root*
44	"P,#,#"		97	FC? 43	*is RAD on?*
45	MAT+	*[#] = [B] + p[I]*	98	GTO 01	*yes, complex roots*
46	CLA		99	X<> Z	*no, real roots*
47	ARCL 01		100	CLX	*so we clear Z*
48	"|-,#,P"		101	X<> Z	
49	M*M	*B= A (B - p I)*	102	LBL 01	
50	ISG 00		103	QROUT	*output roots*
51	GTO 00		104	LBL 00	
52	DIM?	*n,00n*	105	MNAME?	*bring MNAME back*
53	FRC	*0,00n*	106	END	*done*
54	E				

Particular case: Symmetric Matrices { JACOBI } *- Moved to Advntg_MATH Module*

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

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

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

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

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

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

*b*ip = *a*ip cos z – *a*iq sin z
*b*iq = *a*ip sin z + *a*iq cos z
*b*ik = *a*ik ; where i,k # p,q

*b*pp = *a*pp cos^2 z + aqq sin^2 z – 2 apq sin z cos z
*b*qq = *a*pp sin^2 z + *a*qq cos^2 z + 2 *a*pq sinz cos z
*b*pq = 0, and the remaining elements are symmetric.

where: sin z = w / sqrt(2(1+sqrt(1-w^2))), and cos z = sqrt (1-sin^2 z)
with: L = - *a*pq, M = (*a*pp-*a*qq) / 2 , and w = L sign(M) / sqrt (M^2+L^2)

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

Program remarks. The accuracy and running times are display settings-dependent, however the computed eigenvalues are very often more accurate that it'd appear; for instance, using FIX 5 it's quite possible to have eigenvalues accurate to 8 decimal digits. The program is written to be as fast as possible and to occupy the 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	"a 1:1=?"	Data entry starts
25, R/S	"a 1:2=?"	
41, CHS, R/S	"a 1:3=?"	
10, R/S	"a 1:4=?"	
6, CHS, R/S	"a 2:2=?"	Note how the symmetric
68, R/S	"a 2:3=?"	elements are skipped
17, CHS, R/S	"a 2:4=?"	
10, R/S	"a 3:3=?"	
5, R/S	"a 3:4=?"	
3, CHS, R/S	"a 4:4=?"	input the last element
2, R/S	"PREC.=?"	Asks for precision
5, R/S	"RUNNING. . ."	Scrolling on the display
	"X=0,03302"	
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**"	4.004	Creates mtrix in X-Mem
XEQ "**PMTM**"	"R 1: _"	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	"R 3: _"	prompts for row-3
10, ENTER^, CHS, 17, ENTER^,		
5, ENTER^, CHS, 3, R/S	"R 4: _"	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
	"a 1=-35"	Coefficient of x
	"a0= 1,00000"	First coef. (independent term)
	"RUNNING. . . "	Scrolling on the display
	"X 1=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**
with roots shown in the text above.

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:

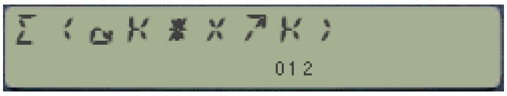
$$\sum_{i=0}^{n} a_i x^i$$

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.

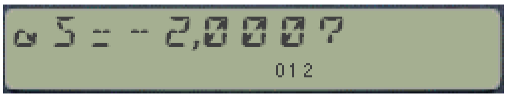
The Polynomial Editor. There are three functions available in the SandMatrix to enter and review polynomials in the calculator. The main one is **PEDIT**, which takes the input from the control word in the X-register and sequentially prompts for each coefficient value. The first thing it does is present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



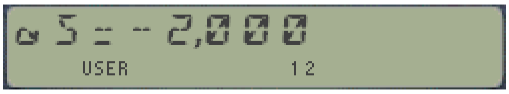
```
Σ(aK*X↑K)
                012
```

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.


a5:-2,000?
012

In edit mode

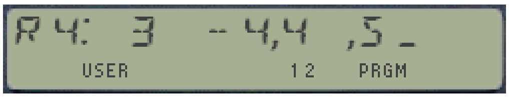

a5:-2,000
USER 12

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.

This is obviously 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.


R4: 3 -4,4 ,5_
USER 12 PRGM

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

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.

<u>Example 1.</u>- Find the result of the <u>polynomial product</u> 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: $p(x) = 4.x^7 + 4.x^6 - 55.x^5 + 114.x^4 - 84.x^3 + 24.x^2 - 12.x + 5$

<u>Example 2.</u>- Find the quotient and reminder for the <u>polynomial division</u> 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:

$a(x) = 2.x^3 + 4.x^2 + 5.x + 6$ and $b(x) = 2.x^3 - 3.x^2 + 7.x + 1$

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

4.2.3. Deleting Coefficients and Moving Polynomials { DTC , PCPY , PSWP }

The functions in this section perform the following housekeeping chores:

- Copy a polynomial from a source to a destination location,
- Swap polynomial coeffiecients between two locations, and
- Delete small coefficients (below 1E-7), which typically appear due to rounding errors in the intermediate operations. This has a cumulative effect that can alter the final result if not corrected.

In revision “N” of the SandMatrix both **PCPY** and **PSWP** are implemented as MCODE functions – faster and more flexible, as they allow overlapped ranges of data registers. **PCPY** is located in the main FAT but **PSWP** is incuded in the secondary FAT, thus you must use one of the launchers to execute it.

These functions require the polynomial control word bbb.eee in Y, and the new 1st. register address in X. The new control word is left in X upon completion. Only the control word in X is required as input for **DTC**.

The three small routines below were written by JM Baillard. 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 the FOCAL program above be careful that the register ranges for both polynomials do not overlap.

4.2.4. Polynomial Evaluation. 1st & 2nd derivatives. { PVAL , dPL , dPL2 }

Calculating derivatives and integrals of polynomial functions is particularly simple. For the polynomial function P(x) on the left below the derivative with respect to x and the indefinite integral are shown in the center and right formulas respectively:

$$\sum_{i=0}^{n} a_i x^i \quad ; \quad \sum_{i=1}^{n} a_i i x^{i-1} \quad ; \quad \sum_{i=0}^{n} \frac{a_i}{i+1} x^{i+1} + c.$$

The subject of Polynomial evaluation is as old as the HP calculators are – surely you remember examples provided in the HP-25 manual and even earlier models. It’s no surprise then that it’s found its way to the SandMatrix, with the added bonus of morphing into an all-MCODE implementation, which has been extended to include the evaluation of the first and second derivatives as well.

The evaluation leaves the result value in X and the argument in LastX – but the control word is maintained in the Y-register for convenience. Let’s see a few examples to clarify:

Example. Evaluate the Polynomial and the first two derivatives for P(x) below at points x=2 and x=-3

P(x) = 5 x^5 + 4 x^4 + 3 x^3 + 2 x^2 + x + 1

Using **PMTP** to enter the coefficients in { R01, R06 }, we type as follows:

```
1.006, XEQ “PMTP”      =>      “R 1:?       “
5, ENTER^, 4, ENTER^, 3, ENTER^, 2, ENTER^, 1, ENTER^, 1,   R/S
```

The control word is left in X, therefore just type the point of evaluation and the required function names as shown below:

```
2, XEQ “PVAL”                =>  259,0000000
RDN, 3, CHS, XEQ “PVAL”      => -956,0000000

RDN, 2, ΣV$ “DPL”            => 573,0000000
RDN, 3, CHS, ΣML  [ , ]      => 1.663,000000

RDN, 2, ΣV$  “DPL2”          =>  1.032,000000
RDN, 3, CHS, ΣML  [ , ]      => -2.318,000000
```

You can *verify the obtained results* using the analytical expressions for P’(x) and P”(x);

P’(x) = 25 x^4 + 16 x^3 + 9 x^2 + 4 x +1
P”(x) = 100 x^3 + 48 x^2 +18 x +4

simply enter their coefficients in memory and use **PVAL** and **dPL** as appropriate. For example, evaluating P’(x) and its first derivative at x=2:

```
7.011, XEQ “PMTP”, {25, ENTER^, 16, ENTER^, 9, ENTER^, 4, ENTER^, 1}, R/S
2,  XEQ “PVAL“               => 573,0000000
RDN, 2, ΣV$ “DPL”            => 1.032,000000
```

And finally, closing the circle of derivatives we enter P”(x) in memory:

```
12.015, XEQ “PMTP”, {100, ENTER^, 48, ENTER^, 18, ENTER^, 4 }, R/S
2,  XEQ “PVAL”               => 1.032,000000
```

Polynomial Integral and Evaluation { ITPL , PVAL }

But wait, there’s more: PVAL can also *evaluate the primitive polynomial,* i.e. calculating the integral between zero and the evaluation point of the original polynomial. Simply change the sign of the control word in Y so it’s negative to trigger this mode.

If you prefer a dedicated function, the entry ITPL, is also available for this purpose that uses the control word in its standard form (i.e. positive value). You can choose which way to use as preferred.

Example 1: Calculate the integral of Q(x) = x^2 + 3x -7 in the interval [0, 2]

First we enter the coefficients in registers { R09 – R11 } using **PMTP** as usual, then we type:

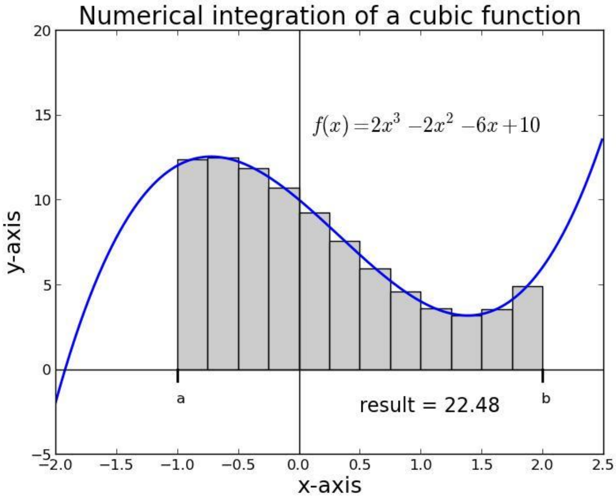
9.011, CHS, 2, **PVAL** => -5,333333334 , or alternatively:
9,011, 2, **ΣV$** “ITPL” => -5,333333334

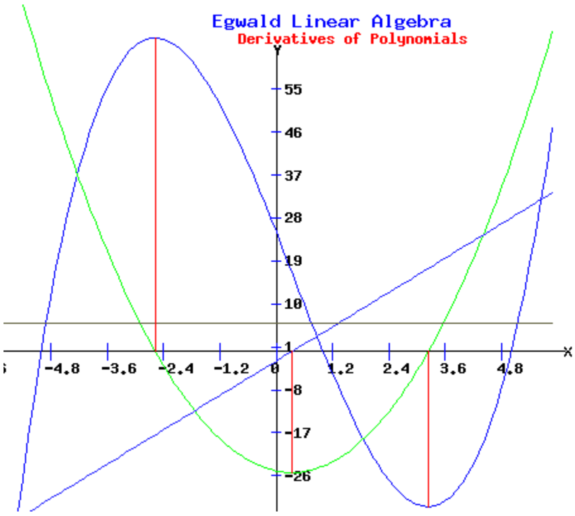
obtained in a blazing fast short time that sure beats the heck out of INTEG / FINTG approach.

Example 2: Calculate the integral between -1 and 2 for P(x) = 2.x^3 – 2.x^2 – 6.x + 10

First we introduce the coefficients in the data registers of choice using **PMTM** as before. Then we’ll modify the integration limits to take advantage if **ITPL**, thus we can write:

ITG[-1, 2] = ITG[0, 2] – ITG[0, -1] = **ITPL**(2) – **ITPL**(-1) = 10,66666667 – (-11,83333333) =
= 22,50000000





The figure on the left shows the polynomial in question, with an approximation done using trapezoidal rule’s rectangles amounting to 22.48 in that example.

The figure on the right shows a book example of a cubic polynomial and its first & second derivatives (parabola and straight line), and even the third derivative (constant line) – here all have real roots but that’s clearly not always the case. Note how the roots of a function correspond to the points where the derivative ot its primitive polynomial is zero, i.e. their local minimum/maximum – as we know well from function theory.

Appendix.- Polynomial Real Roots

There’s much to be said about simple functions like these, that when implemented in MCODE provide maximum speed and convenience. As an exercise, you’re encouraged to write a polynomial **real** root-finder routine using Newton’s (left) or Halley’s (right) methods for the successive approximations to the roots. Programming them for polynomials should become a near-trivial task using the functions described above.

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

The program below is taken from JM Baillard’s web pages, a perfect example of the applicability of the evaluation functions. It solves the equation p(x) = a0.x^n + a1.x^n-1 + ... + an-1.x + an = 0, provided all roots are real.

The coefficients { a0 , a1 , ... , an-1 , an } are to be stored into contiguous registers Rbb thru Ree. Starting with an initial guess x0 , the Newton's formula is applied until | p(xk)/p'(xk) | is smaller than 10-9 (line 24). Then, p is replaced by p/(x-r) (lines 32 to 39) and the root is stored into Ree.

The process is repeated until all roots are found (polynomial deflation). Finally, the roots are in registers {Rbb+1 , ... , Ree}, (i.e. the original coefficients are overwritten) - and the control word is left in X upon completion.

```
01  LBL "PLR"
02  STO 01
03  X<>Y
04  STO 00
05  STO 03
06  STO 04
07  ISG 04
08  LBL 01
09  VIEW 01
10  RCL 03
11  RCL 01
12  dPL
13  STO 02
14  X<> L
15  PVAL
16  RCL 02
17  /
18  ST- 01
19  RCL 01
20  X=0?
21  SIGN
22  /
23  ABS
24  E-9
25  X<Y?
26  GTO 01
27  E-3
28  ST- 03
29  RCL 03
30  STO 02
31  CLX
32  LBL 02
33  RCL 01
34  *
35  ST+ IND 02
36  RCL IND 02
37  ISG 02
38  GTO 02
39  RCL 01
40  STO IND 02
41  ISG 04
42  GTO 01
43  RCL 00
44  E
45  +
46  CLD
47  END
```

Example: Find all the roots of P(x) = 2.x^5 + 3.x^4 - 35.x^3 - 10.x^2 + 128.x - 74

For example storing the coefficients in {R05 – R10} and if we choose x0 = 1

```
5.010  ENTER^,  1,  XEQ "PLR"     ->    the successive approximations are displayed
                      and finally:   =>    6.010
```

the control number of the solutions in {R06 - R10}

```
RCL 06  =>  -4.373739462
RCL 07  =>  -2.455070118
RCL 08  =>   2.984066207
RCL 09  =>   1.641131729
RCL 10  =>   0.703611645
```

For more examples and other programs to cover non-real roots see Jean’Marc’s pages located at: http://hp41programs.yolasite.com/polynomials.php

4.2.5. Polynomials over Integer field: Cohn’s irreducibility criterion.

If F is a field, a non-constant polynomial is irreducible over F if its coefficients belong to F and it cannot be factored into the product of two non-constant polynomials with coefficients in F.

Arthur Cohn's irreducibility criterion is a sufficient condition for a polynomial to be irreducible in **Z[x]** — that is, for it to be unfactorable into the product of lower-degree polynomials with integer coefficients.

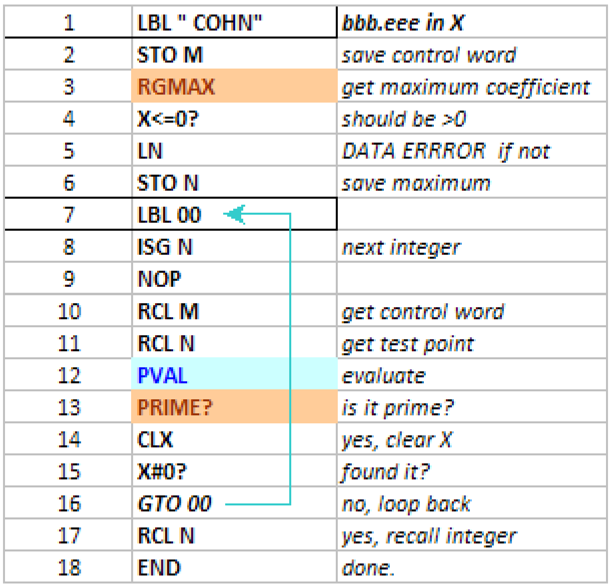
The criterion is often stated as follows: If a prime number p is expressed in base 10 as

$$p = a_m 10^m + a_{m-1} 10^{m-1} + \cdots + a_1 10 + a_0$$

where 0<= ak < 9, then the polynomial formed with the same factors below is irreducible in Z[x]:

$$f(x) = a_m x^m + a_{m-1} x^{m-1} + \cdots + a_1 x + a_0$$

?IRR is a peculiar function, in that the irreducibility is tested by “brute force”, using a sequence of integers looking for a prime number result. If found, the polynomial is irreducible, but if not found within the range of the calculator then there’s no conclusive determination. So this is a glorified example of the trial and errortechnique, so to speak.



1	LBL " COHN"	*bbb.eee in X*
2	STO M	*save control word*
3	RGMAX	*get maximum coefficient*
4	X<=0?	*should be >0*
5	LN	*DATA ERRROR if not*
6	STO N	*save maximum*
7	LBL 00	
8	ISG N	*next integer*
9	NOP	
10	RCL M	*get control word*
11	RCL N	*get test point*
12	PVAL	*evaluate*
13	PRIME?	*is it prime?*
14	CLX	*yes, clear X*
15	X#0?	*found it?*
16	*GTO 00*	*no, loop back*
17	RCL N	*yes, recall integer*
18	END	*done.*

Using functions from the SandMath makes programming this criterium a simple affair, see the program listing on the left with functions **RGMAX** and **PVAL** doing all the heavy-lifting for you, and of course function **PRIME?** finally testing the results for primality.

A restriction of this method is that all coefficients must be positive, and obviously the independent term a0 not null.

Example1: $p(x) = 2x^4 + 3x^2 + 5x + 1$,
?IRR returns n=6, which result p(6) = 2,731 is a prime

Example2: $p(x) = 2x^6 + 3x^5 + 4x^3 + 5x^2 + 2x + 1$,
?IRR returns n=8, which result p(8) = 624,977 is a prime

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:

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

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 the Alpha for data storage, but no data registers at all.

The program does the data output automatically, 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 shows "J" to denote the imaginary unit "i"

Example 1: 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
```

Example 2: 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

X3= 3,000, X4= 6,000

In this and next example, note how true integer values (up to the 9th. digit) don’t display zeros after the decimal point, a much clearer display with added informational value.

Example 3: 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

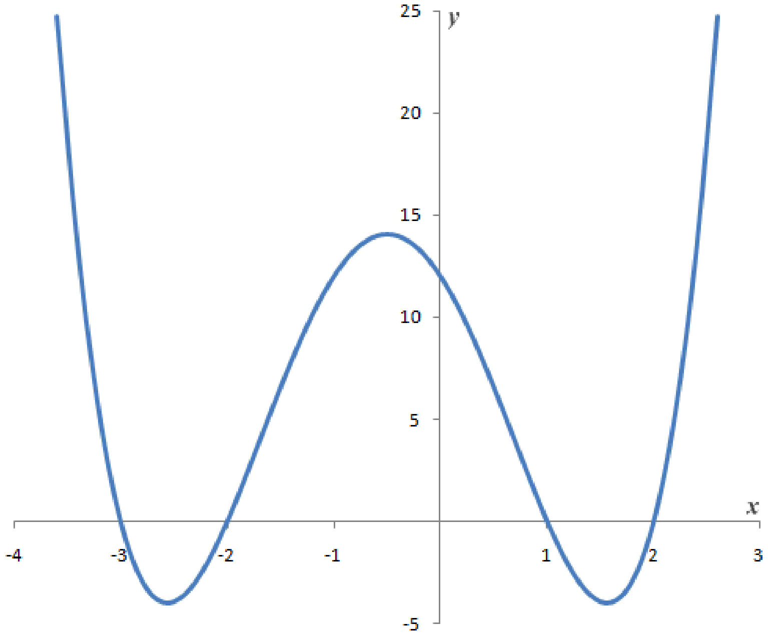
Z= 6 + J3,000

Example 4: Calculate the roots of the polynomial: $Q(x) = x^4 + 2 x^3 - 7x^2 - 8 x + 12$ which plot is shown below.

2, ENTER^, 7, CHS, ENTER^, 8, CHS, ENTER^, 12, XEQ “**QUART**“ >>>>

X1= 2,000; x2= 1,000

X3= -2,000; x4= -3,000



Example 5: Solve $x^4 + x^3 + x^2 + x + 1 = 0$

Simply type: 1, ENTER^, ENTER^, ENTER^, $\boxed{\text{QUART}}$, >>>

Z= 0.309 + J0.951 , R/S

Z= -0.809 + J0.588

4.3.2. General case: degree N. { PROOT , 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 finding Q(x) - one quotient polynomial of degree two, plus R(x), 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)]

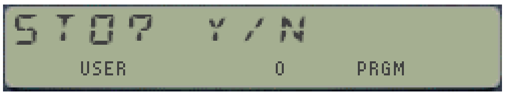
The program dimensions and populates matrices [**RS**] and [**CN**] to hold the current values of p,q and the coefficients Cn respectively:

- [**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:


STOP Y/N

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 **PROOT** is also meant to be used in subroutines for program execution. 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	“a 1= ?”	
190, R/S	“a0= ?	
150, R/S	“STOP 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:

```
                 [[ -2     1  ]
                  [ -2    -1  ]
[ROOTS ] =        [  1     3  ]
                  [  1    -3  ]
                  [ -1.5   0  ]]
```

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.

Program listing.-

```
 1  *LBL “PROOT”
 2  RAD
 3  ΣF#          ?RTN
 4  GTO 00
 5  SF 01
 6  SIZE?
 7  ?ORDER
 8  STOP
 9  INT
10  ABS
11  X=0?         Zero?
12  1/X`         abort
13  STO Z
14  ST+ X
15  9
16  +
17  X>Y?
18  PSIZE
19  RCL Z
20  STO 00
21  3
22  +
23  E3/E+
24  2
25  +
26  PEDIT
27  CF 21
28  "STO? Y/N"
29  AVIEW
30  ΣF#
31  94           Y/N?
32  CLX
33  X#0?
34  GTO 00
35  GTO 07
36  *LBL “RTS”
37  CF 01
38  *LBL 07
39  RCL 00
40  2
41  I<>J
42  +
43  "ROOTS"
44  MATDIM
45  PURFL
46  MATDIM
47  ,
48  MSIJA
49  *LBL 00
50  RUNNING
51  "CN"
52   E
53  E3/E+
54  ST+ X
55  MATDIM
56  "RS"
57  INT
58  MATDIM
59  *LBL 97
60  2
61  RCL 00
62  X=Y?
63  GTO 92
64  X=1?
65  GTO 91
66  ,5
67  STO 01
68  STO 02
69  *LBL 98
70  RCL 00
71  4
72  +
73  SIZE?
74   E
75  -
76  I<>J
77  +
78  CLRGX
79  INT
80  RCL X
81  RCL 00
82  +
83  I<>J
84  +
85  STO M
86  3.1
87  STO N
88  *LBL 09
89  RCL M
90  RCL IND X
91  RCL 02
92  *
93  ISG Y
94  ""
95  RCL IND Y
96  RCL 01
97  *
98  +
99  CHS
100 RCL IND N
101 +
102 ISG Y
```

```
103  ""
104  STO IND Y
105  ISG N
106  ISG M
107  GTO 09
108  "RS"
109  .
110  MSIJA
111  RCL 00
112  ST+ X
113  6
114  +
115  RCL IND X
116  CHS
117  DSE Y
118  RCL IND Y
119  CHS
120  MSC+
121  RCL 01
122  *
123  +
124  MSC+
125  6
126  RCL 00
127  +
128  LASTX
129  ST+ X
130  4
131  +
132  I<>J
133  +
134  STO M
135  RCL 00
136  ST+ X
137  7.1
138  +
139  STO N
140  LBL 10
141  RCL N
142  RCL IND X
143  RCL 02
144  *
145  ISG Y
146  ""
147  RCL IND Y
148  STO IND N
149  RCL 01
150  *
151  +
152  RCL IND M
153  +
154  CHS
155  STO IND Y
156  ISG M
157  GTO 10
158  "CN"
159  ,
160  MSIJA
161  RDN
162  MSR+
163  DSE Y
164  ""
165  RCL IND Y
166  MSR+
167  RCL 02
168  CHS
169  *
170  MSR+
171  LASTX
172  /
173  RCL 01
174  *
175  +
176  MSR+
177  >",RS"
178  MSYS
179  MNAME?
180  ,
181  MSIJA
182  MRR+
183  ST+ 01
184  MRR+
185  ST+ 02
186  SUMAB
187  RCL 01
188  ABS
189  RCL 02
190  ABS
191  +
192  /
193  VIEW X
194  RND
195  X#0?
196  GTO 98
197   E
198  RCL 01
199  RCL 02
200  TONE 2
201  XEQ 02
202  RCL 00
203  5
204  +
205  3
206  E3/E+
207  +
208  RCL 00
209   E
210  -
211   E6
212  /
213  +
214  REGMOVE
215  2
216  ST- 00
217  GTO 97
218  *LBL 92
219  RCL 03
220  RCL 04
221  RCL 05
222  XEQ 02
223  FC?C 01
224  RTN
225  GTO J
226  *LBL 02
227  QROOT
228  FS? 01
229  GTO 08
230  QROUT
231  STOP
232  RTN
233  *LBL 08
234  CF 02
235  RCL Z
236  X=0?
237  SF 02
238  RDN
239  "ROOTS"
240  MRIJA
241  RDN
242  MSC+
243  FS?C 02
244  GTO 00
245  X<>Y
246  MSC+
247  RTN
248  *LBL 00
249  MSR+
250  I-
251  X<>Y
252  MSC+
253  CHS
254  MSR+
255  RTN
256  *LBL 91
257  RCL 04
258  RCL 03
259  /
260  CHS
261  "ROOT="
262  FS? 01
263  GTO 00
264  ARCL X
265  PROMPT
266  RTN
267  *LBL 00
268  ΣV#
269  25              ABSP
270  >"S"
271  MRIJA
272  RDN
273  MSC+
274  *LBL J
275   ,
276  MSIJ
277  *LBL 01
278  .
279  278 MRR+
280  MRR+
281  CF 09
282   X#0?
283   I+
284   X<>Y
285   QROUT
286   STOP
287   FS? 09
288   RTN
289   FC? 10
290   GTO 01
291   END
```

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. The error message "bbb<=8" is shown otherwise. For both programs the accuracy of the solutions (and therefore their run times) depends on the display settings.

BAIRS factorizes the polynomial
$p(x) = a_0.x^n+a_1.x^{n-1}+ \dots + a_{n-1}.x+a_n$ 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**"	"R 9: _ "	
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"

Program listing.-

```
01 *LBL “BAIRS”
02 RAD
03 CF 21
04*LBL 00
05 9
06 X<>Y
07 X>Y?
08 GTO 00
09 "bbb<=8"
10 PROMPT
11 GTO 00
12*LBL 00
13 STO 04
14 STO 08
15 2
16 STO 06
17 PI
18 STO 07
19*LBL 01
20 CLX
21 STO 00
22 STO 01
23 STO 02
24 STO 03
25 RCL 04
26 STO 05
27*LBL 02
28 RCL IND 05
29 RCL 00
30 RCL 07
31 *
32 -
33 RCL 01
34 RCL 06
35 *
36 -
37 X<> 01
38 STO 00
39 RCL 02
40 RCL 07
41 *
42 -
43 RCL 03
44 RCL 06
45 *
46 -
47 X<> 03
48 X<> 02
49 ISG 05
50 GTO 02
51 STO 05
52 RCL 01
53 *
54 RCL 00
55 RCL 02
56 ST* 01
57 *
58 -
59 RCL 03
60 RCL 05
61 *
62 RCL 02
63 X^2
64 -
65 STO 05
66 /
67 ST+ 06
68 RCL 00
69 RCL 03
70 *
71 RCL 01
72 -
73 RCL 05
74 /
75 ST+ 07
76 R-P
77 RND
78 VIEW 06
79 X#0?
80 GTO 01
81 SIGN
82 STO 05
83 RCL 04
84 7
85 I<>J
86 5
87 +
88 PDIV
89 STO 04
90 RCL 06
91 STO IND Z
92 ISG Z
93 RCL 07
94 STO IND T
95 RCL 04
96 2
97 +
98 ISG X
99 GTO 01
100 CLD
101 RCL 08
102*LBL E
103 RCL 08
104 STO 04
105 ΣF#
106 93
107 ODD?
108 GTO 03
109 RCL IND 04
110 ISG 04
111 GTO 05
112*LBL 03
113 .
114 RCL IND 04
115 ISG 04
116 RCL IND 04
117 X<>Y
118 /
119 CHS
120 TONE 9
121 ΣF#
122 32
123 STOP
124 ISG 04
125*LBL 04
126  E
127*LBL 05
128 RCL IND 04
129 ISG 04
130 RCL IND 04
131 QROOT
132 ISG 04
133 FS? 30
134 GTO 06
135 TONE 9
136 QROUT
137 STOP
138 GTO 04
139*LBL 06
140 BEEP
141 QROUT
142 STOP
143 END
```

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
0	**EQT**	Curve Equation Display	Equation number in X (0 to 15)
1	**POLINT**	Polynomial interpolation	Under program control
2	**PRMF**	Prime Factors decomposition	Argument in X-reg
3	**PF>X**	From prime factors to argument	Prime factors in matrix [PRMF]
4	**TOTNT**	Euler's Totient function	Argument in X-reg
5	**POLFIT**	Polynomial Fitting	Under program control
6	**POLZER**	From Poles to Zeroes	Under program control
7	**PFE**	Partial Fractions Expansion	Under program control

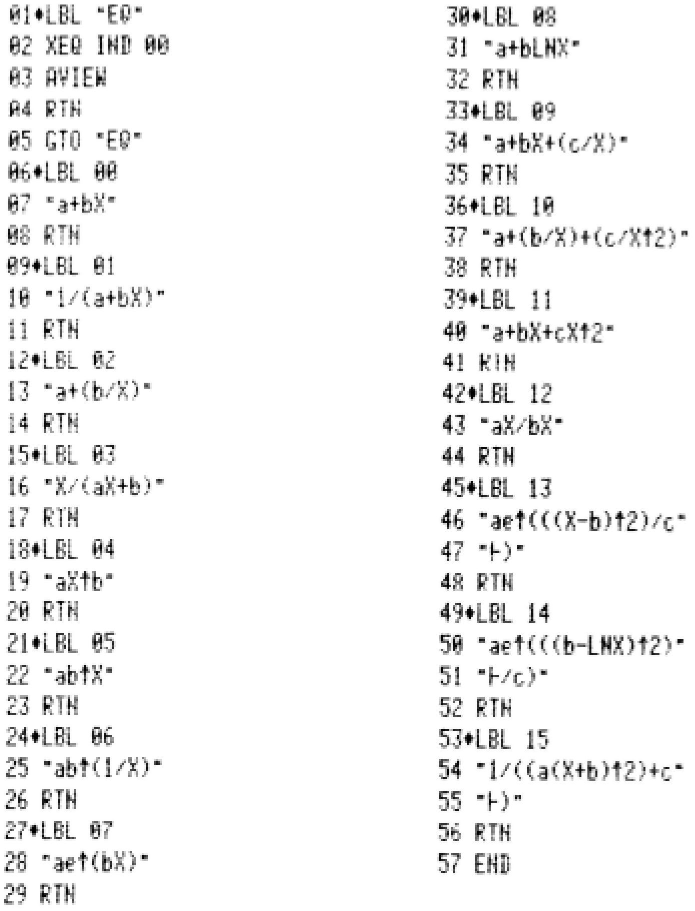
4.4.1. Displaying the Equations for Curve Fitting Programs { EQT }

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 Hyperbolic
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 FOCAL program 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
```

The implementation in the SandMatrix is done in MCODE, much longer in size (about 350 bytes in total) but possible to tuck away in a second bank – where the space for it was available.

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	Comment
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

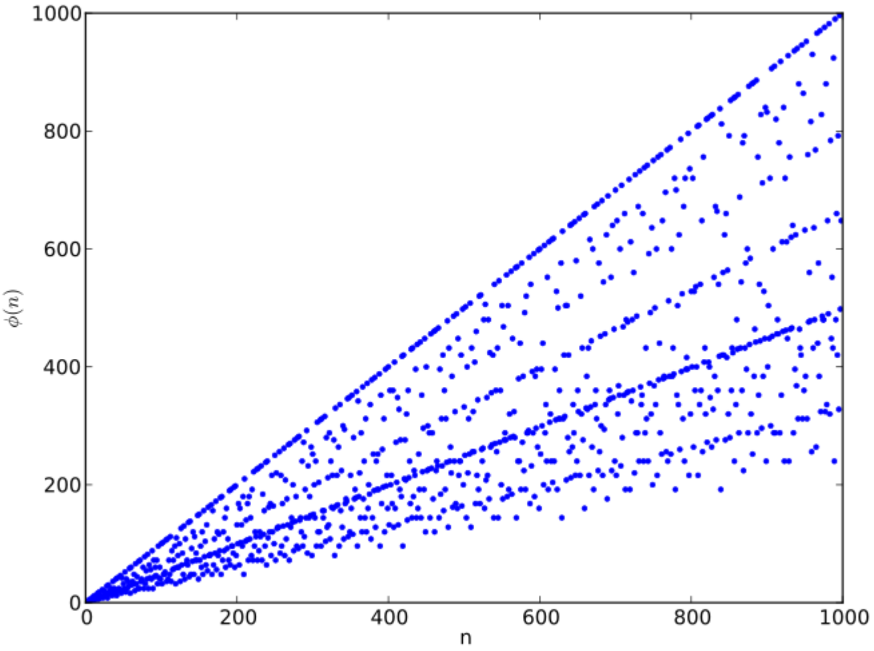
The first one PRMF extends the basic prime factorization capability in the SandMath, **PFCT**. The difference is that whereas **PFCT** only uses the Alpha register to output the result (as Alpha string), here the prime factors and their multiplicities are also stored in a matrix in X-Mem - named [**PRFM**]. 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 = ∏ 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:

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

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	"X 1=?"
- 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	"Y 1=?"
- 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	"a 1=- 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	[EM Room left - this value varies with your configuration]

4.4.5. Orthogonal Polynomials.

Hermite Polynomials. { HMT }

The Hermite Polynomials were defined by Laplace (1.810) though in scarcely recognizable form, and studied in detail by Chebyshev (1.859). Chebyshev's work was overlooked and they were named later after Charles Hermite who wrote on the polynomials in 1.864 describing them as new.

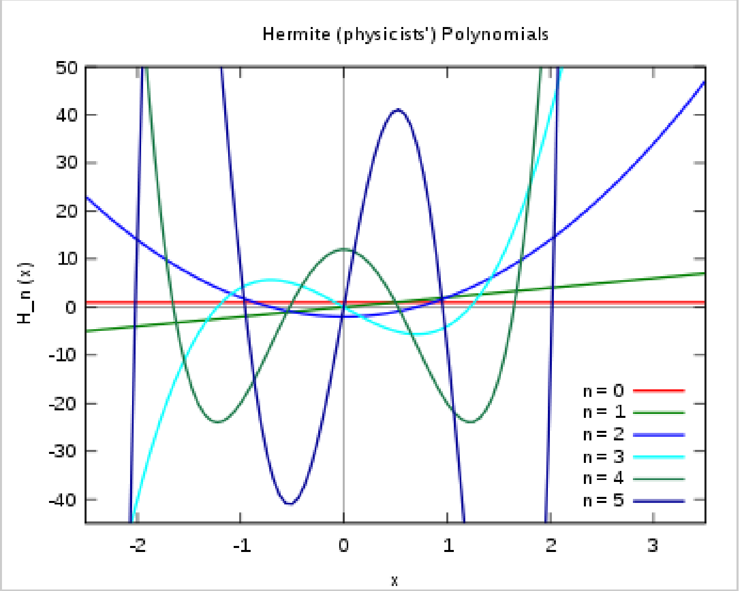
There are two different ways of standardizing the Hermite polynomials; The "probabilists' [He(n)] and the “physicists’ [H(n)] Hermite polynomials" given by the following expressions:

$$He_n(x) = (-1)^n e^{\frac{x^2}{2}} \frac{d^n}{dx^n} e^{-\frac{x^2}{2}} \qquad H_n(x) = (-1)^n e^{x^2} \frac{d^n}{dx^n} e^{-x^2}$$

These two definitions are not exactly identical; each one is a rescaling of the other,

$$H_n(x) = 2^{\frac{n}{2}} He_n(\sqrt{2}\,x), \qquad He_n(x) = 2^{-\frac{n}{2}} H_n\left(\frac{x}{\sqrt{2}}\right)$$

The figure below shows the first six Hermite polynomials Hn, which are those implemented in the SandMatrix using the recurrence expression: $H_{n+1}(x) = 2xH_n(x) - 2nH_{n-1}(x).$



The first six (physicists') Hermite polynomials $H_n(x)$.

$$\begin{aligned} H_0(x) &= 1 \\ H_1(x) &= 2x \\ H_2(x) &= 4x^2 - 2 \\ H_3(x) &= 8x^3 - 12x \\ H_4(x) &= 16x^4 - 48x^2 + 12 \\ H_5(x) &= 32x^5 - 160x^3 + 120x \\ H_6(x) &= 64x^6 - 480x^4 + 720x^2 - 120 \end{aligned}$$

They are written as fully MCODE functions, thus the execution time is optimized – wich is especially relevant given the recurrent nature of the method employed.

Examples. Calculate H7(3.14) and H3(-2.5)

we type: 7, ENTER^, 3.14, **ΣV$** “HMT” => 73,726.24325
and: 3, ENTER^, 2.5, CHS, **ΣML**, [,] => -95.00000000

Legendre Polynomials { LEG }

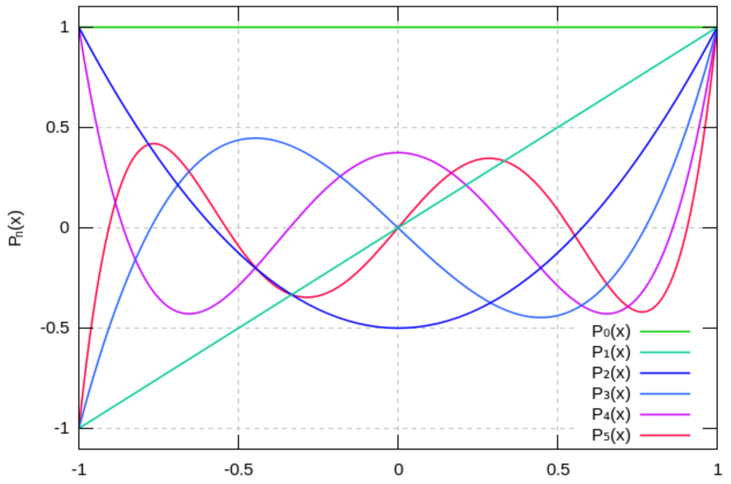
In mathematics, Legendre functions are solutions to Legendre's differential equation shown below. They are named after Adrien-Marie Legendre. This ordinary differential equation is frequently encountered in physics and other technical fields. In particular, it occurs when solving Laplace's equation (and related partial differential equations) in spherical coordinates.

$$\frac{d}{dx}\left[(1-x^2)\frac{d}{dx}P_n(x)\right]+n(n+1)P_n(x)=0.$$

These solutions for n = 0, 1, 2, ... (with the normalization Pn(1) = 1) form a polynomial sequence of orthogonal polynomials called the Legendre polynomials. Each Legendre polynomial Pn(x) is an nth-degree polynomial.

$$P_n(x)=\frac{1}{2^n n!}\frac{d^n}{dx^n}\left[(x^2-1)^n\right]$$

The figure below shows the first six Legendre polynomials, with their explicit expressions on the left:



$$P_n(x)$$

$$1$$

$$x$$

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

$$\tfrac{1}{2}(5x^3-3x)$$

$$\tfrac{1}{8}(35x^4-30x^2+3)$$

$$\tfrac{1}{8}(63x^5-70x^3+15x)$$

$$\tfrac{1}{16}(231x^6-315x^4+105x^2-5)$$

Like in the Hermite's case, the SandMatrix implementation (also written in MCODE for better performance and accuracy) uses a recursive approach to calculate them, described by the Bonnet's formula below:

$$(n+1)P_{n+1}(x)=(2n+1)xP_n(x)-nP_{n-1}(x).$$

Examples: Calculate L7(4.9), and L3(-2.5)

we type:	7, ENTER^, 4.9, **ΣV$** "LEG"	=> 1,698,444.018
and:	3, ENTER^, 2.5, CHS, **ΣML**, [,]	=> -35.31250000

Note: Remember that the SandMatrix includes functions **CHB1** and **CHB2** to calculate the Chebyshev polynomials; you can refer to the SandMatrix Manual for details. The implementation there is also done in MCODE, thus as fast and convenient as possible.

Note that despite being sub-functions from the SandMath, both CHB1 and CHB2 are available from the **ΣML** "P: _" launcher as well. – and as such they'll be logged in its own LASTF buffer when used.

Laguerre and Associated Laguerre Polynomials. { LAG , LANX }

In mathematics, the Laguerre polynomials, named after Edmond Laguerre (1834 - 1886), are solutions of Laguerre's equation - which is a second-order linear differential equation. This equation has nonsingular solutions only if n is a non-negative integer.

$$xy'' + (1-x)y' + ny = 0$$

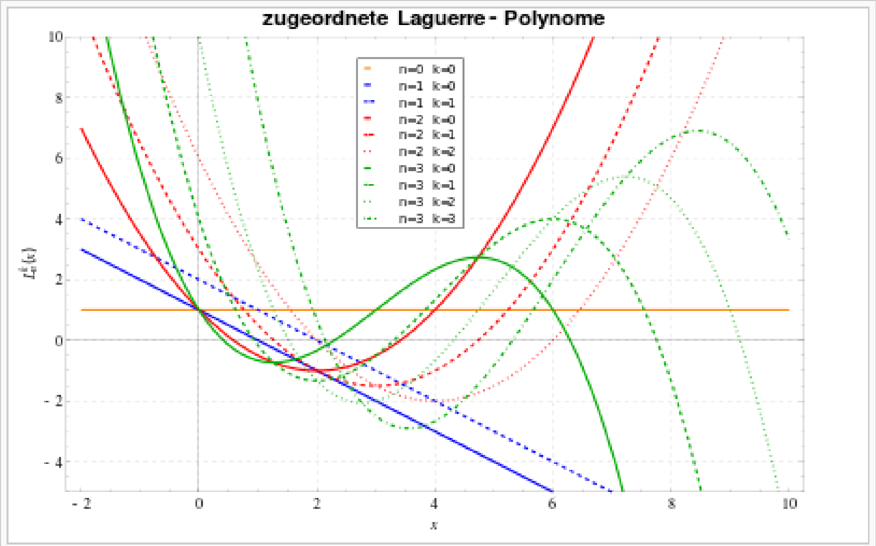
The Associated Laguerre polynomials (alternatively, but rarely, named Sonin polynomials after their inventor, Nikolay Yakovlevich Sonin) are solutions of

$$xy'' + (\alpha + 1 - x)y' + ny = 0$$

One can also define the Laguerre polynomials recursively, defining the first two polynomials as shown below (where α=0 for the standard type), and then using the following recurrence relation, for any term k ≥ 1:

$$L_0^{(\alpha)}(x) = 1$$
$$L_1^{(\alpha)}(x) = 1 + \alpha - x$$

$$L_{k+1}^{(\alpha)}(x) = \frac{(2k+1+\alpha-x)L_k^{(\alpha)}(x) - (k+\alpha)L_{k-1}^{(\alpha)}(x)}{k+1}.$$



The first few associated Laguerre polynomials

Examples: Calculate **L7**(3.14) and **L7**(1.4, 3.14)

The input parameters are entered similarly in the stack for both case, with "a" in the Z- register for the generalized case, and "n" and "x" in the Y- and X-registers in both instances.

```
        7, ENTER^, 3.14, ΣV$ "LAG"                  =>  -0.978658720
and:
        1.4, ENTER^, 7, ENTER^, 3.14, ΣV$ "LANX"    =>   1.692567095
```

As a reminder, the SandMath module includes **LAYX**, another generalization of the Laguerre polynomias for non-integer orders. It is a direct application of the Kumer M function; refer to the SandMath manual for details.

4.4.6. Orthogonal polynomial Fit. { OPFIT } *- Moved to Curve_Fitting Module*

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 Σ Pi 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

Credits: The original version (labeled “**APOLO**”) was written by Eugenio Úbeda, and published in the UPLE. The version in the SandMatrix only had minimal changes made to it. It was by far the longest program in the module, and unfortunately had to be removed to make room for the Vector Functions.

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

These programs complete the applications section. The first two calculate the coefficients and 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 third program does the reverse, i.e. it 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	**POLZR**	Coefficients of "Natural" Polyn	Under program control
2	**PFE**	Partial Fraction Expansion	Under program control

(*) *POLZER ha been - Moved to Advntg_MATH Module*

These programs calculate 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), whereas **POLZR** is not. The caveat is that **POLZR** will only calculate the coefficients of the natural polynomial, but not its roorts. You can of course use **PROOT** or **BAIRS** manually for that purpose.

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
R/S	a(3)=-40
R/S	a(2)= 105
R/S	a(1)=- 100
R/S	a(0)=24

Therefore the “natural” polynomial form is as follows:

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

This is where **POLZR** lets thing go, leaving the polynomial control word bbb.eee in the X-Register. For **POLZER**’s case however, the execution is next transferred to **PROOT** (or to **QUART** if #p=5) which calculates the roots following the iterative process explained in section 4.3.1. Remember that the accuracy is dictated by the number of decimals places set .

```
R/S             "RUNNING. "
                X 1= 3,6
R/S             X2=2,54395
R/S             X3= 1,45609
R/S             X4=0,35557
```

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

4.4.7. Partial Fraction Decomposition { PFE }

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:

$$\begin{aligned} R(x) = E(x) \;&+\; p_{1,1}(x) / [\, q_1(x)\,]^{\mu 1} + p_{1,2}(x) / [\, q_1(x)\,]^{\mu 1\text{-}1} + \dots\dots\dots + p_{1,\mu 1}(x) / q_1(x) \\ &+ \dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots \\ &+ p_{n,1}(x) / [\, q_n(x)\,]^{\mu n} + p_{n,2}(x) / [\, q_n(x)\,]^{\mu n\text{-}1} + \dots\dots\dots\dots + p_{n,\mu n}(x) / q_n(x) \end{aligned}$$

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	“a 1=?”	
7, R/S	“a0=?”	
10, R/S	“D 1#=?”	Input degree of Q1 in den.
2, R/S	“Σ(aK*X^K)”	Reminder of convention
	“a2=?”	
2, R/S	“a 1=?”	
1, R/S	“a0=?”	
1, R/S	“D2#=?”	
1, R/S	“Σ(aK*X^K)”	Reminder of convention
	“a 1=?”	
1, R/S	“a0=?	
2, CHS, R/S	“XP^μ”	time to enter the multiplicities now
	“a 1= ?”	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
	“a 1=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.

3. 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 ....

4. 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)

<u>Example 2.</u>- 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:

R06: 18.021 with: R18=-2/3, R19= 0, R10 =-1/9, R21 =0
R05: 28.031 with R29=1/9, R29=0, | R30=-2/9, and R31=0
R15: 16,017 with: R16 = 1/9 and R17 = 0

The range in R05 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.

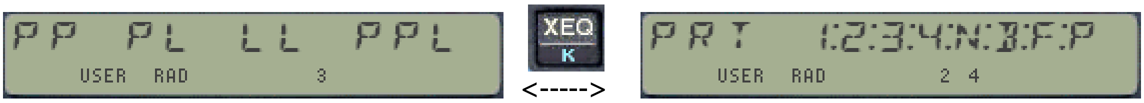
<u>Note:</u>- you can check another Partial Fraction expansion program (by Narmwon Kim) available at the HP-41 archive site, which features a simpler user interaction and data entry/output, but at the expense of more limited functionality. It is also less general-purpose, and more geared towards control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

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

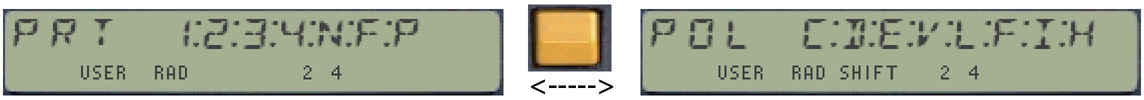
Other Polynomial Launchers – Roots and Utils.

For convenience purposes several root finding and other polynomial utilities functions are grouped in an independent launcher – combining functions from both the SandMath and the SandMatrix, under common themes.

This launcher is accessed as an alternate “personality” of the **ΣDST** (*) function (no more FAT entries were available), pressing the XEQ key to invoke it. In fact, XEQ toggles between both “personalities”, as shown in the picture below:



Like **ΣDST**, the polynomial launcher itself also has two “screens”, toggled with the SHIFT key:



The following tables show which functions are available under each of the two screens. Note that some are from the SandMath module, in a logical grouping of the roots theme.

#	Function	Description	Input
1	**STLINE**	Straight Line equation	{ y1, x1, y2, x2 } in stack
2	**QROOT**	Quadratic Equation	{ a, b, c } in stack
3	**CROOT**	Cubic Equation	{ a, b, c, d } in stack
4	**QUART**	Quartic Equation (monic)	{ b, c, d, e } in stack (a=1)
N	**PROOT**	Polynomial Roots	Follow program instructions
B	**BAIRS**	Bairstow Method	Follow program instructions
F	**FROOT**	Roots of f(x)	FNAME in ALPHA, a, b in Y- / X-reg
P	**POLZER**	From Poles to Zeros	Prompts for poles

#	Function	Description	Input
C	**PCPY**	Polynomial Copy	From / to control words in stack
D	**DTC**	Deletion of Tiny coeffs	Control word in X
E	**PEDIT**	Polynomial Editor	Control word in X
V	**PVIEW**	Polynomial View	Control word in X
L	**PVAL**	Polynomial Evaluation	Control word in Y, point in X
F	**POLFIT**	Polynomial Fitting	Follow program instructions
I	**POLINT**	Polynomial interpolation	Follow program instructions
H	**CHRPOL**	Characteristic polynomial	MNAME in ALPHA

Notes:-

- The back arrow key cancels the action and returns to the main **ΣML** launcher (“M:_”prompt).
- The USER and ALPHA keys are also active to invoke the sub-function launchers **ΣV#** and **ΣF$**.
- You can refer to the appropriate section within this manual or the SandMath manual for specific details of all these functions.
- The SandMath functions executed from here will also be registerer in the SandMatrix’s “Last Function” facility.

*(*) **ΣDST** will be described in the next section of the manual.*

The "Last Function" functionality.

Like the SandMath and other advanced modules, the SandMatrix include support for the "LASTF" functionality. This is a handy choice for *repeat executions of the same function* (i.e. to re-execute the last-executed function), without having to type its name or navigate the different launchers to access it.

The implementation is not universal – it only covers functions invoked using the dedicated launchers, but not those called using the mainframe XEQ function. It does however support two scenarios:

- functions in the main FATs, from ***any*** plugged-in module; as well as
- those *sub-functions from the auxiliary FATs*.

Because the latter group cannot be assigned to a key in the user keyboard, the LASTF solution is especially useful in this case. The following table summarizes the launchers that have this feature:

Module	Launchers	LASTF Method
SandMatrix 4	**ΣML**, **ΣDST**	Captures sub/fnc id#
revision "M"	**ΣV$ _**	Captures sub/fnc NAME
	ΣV# _ _ _	Captures sub/fnc id#
Revision "N"	**CAT+ (XEQ')**	Captures sub/fnc id#

Note that the Alphabetical launcher **ΣV$** will *switch to ALPHA mode automatically*. Spelling the function name is terminated pressing ALPHA, which will either execute the function (in RUN mode) or enter it using *two* program steps in PRGM mode by means of the **ΣV#** function plus the corresponding index (using the so-called non-merged approach). This conversion happens entirely automatically.

With revision "N", the LASTF operation is also supported when excuting a sub-function from within the **CAT+** enumeration, using the [XEQ] hot-key - very handy for those fncs with elusive spelling.

Another new enhancement is the display of the sub-function names when using the index-based launcher **ΣV#** - which provides visual feedback that the chosen function is the intended one (or not). This feature is active in RUN mode, when entering it into a program, and when single-stepping a program execution - but obviously not so during the standard program runs.

LASTF Operating Instructions

No separate function exists. - The Last Function feature is triggered by pressing the radix key (decimal point - the same key used by LastX) while the launcher prompts are up. This is consistently implemented across all launchers supporting the functionality in the three modules (SandMath, SandMatrix and PowerCL) – they all work the same way.

When this feature is invoked, it first briefly shows "LASTF" in the display, quickly followed by the last-function name. Keeping the key depressed for a while shows "NULL" and cancels the action. In RUN mode the function is executed, and in PRGM mode it's added as a program step if programmable, or directly executed if not programmable.

The functionality is a two-step process: a first one to capture the function id#, and a second that retrieves it, shows the function name, and finally parses it. All launchers have been enhanced to store the appropriate function information (either index codes or full names) in registers within a dedicated buffer (with id# = 9). The buffer is maintained automatically by the modules (created if not present when the calculator is 'switched ON), and its contents are preserved while it is turned off (during "deep sleep"). No user interaction is required.

If no last-function information yet exists, the error message "NO LASTF" is shown. If the buffer #9 is not present, the error message is "NO BUF" instead.

5. Vectors and Geometry

5.1. -VECTOR CALC

This section was first inspired by the Advantage Pac FOCAL programs **VC** and **TR** - which have now been re-written using the new MCODE functions for enhanced performance and accuracy.

Where the SandMatrix implementation adds significant value is in the convenience and usability aspects. The vector calculator for instance is a launcher-based scheme that extends the Advantage menu-driven concept a few notches, providing more functionality and wider range – as will be described later on. Data registers are also handled in a vector-oriented fashion, operating on 3-registers blocks at once – and supporting INDirect addressing; all fully programmable using the so-called “non-merged functions” technique.

Accessing the Vector sub-Functions.

Multi-Functions **ΣV#** and **ΣV$** provide access to the entire set of vector sub-functions, grouped in two sections. The sub-functions can be invoked either by its index within the group, using **ΣV#**, or its direct name, using **ΣV$.** This is implemented in such a way that they are also programmable, and can be entered into a program line using a technique called “*non-merged functions*”.

This approach is identical to the one used in the SandMath module – and derives from the original implementation in the HEPAX module.

A sub-function catalog **CAT+** is also available, listing the functions included within the groups. Direct execution (or programming if in PRGM mode) is possible just by stopping the catalog at a certain entry and pressing the XEQ key. The **CAT+** catalog behaves very much live the native ones in the machine: you can stop it using R/S, SST/BST them, press ENTER^ to move to the next “sub-section”, cancel or resume the listing at any time.

As additional bonus, the sub-function launcher **ΣV$** will also search the main FAT if the sub-function name is not found within the multi-function group – so the user needn’t remember where a specific function sought for was located. In fact, **ΣV$** will also find a function from any plugged-in module in the system, even outside of the SandMatrix module.

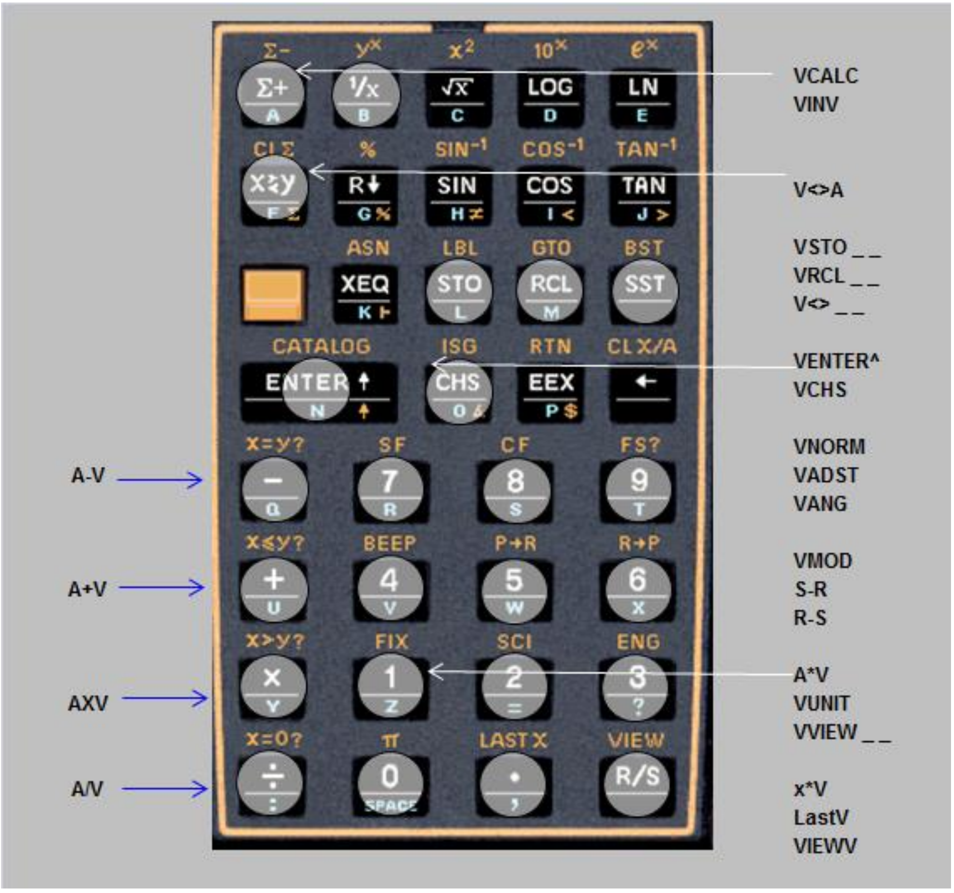
A comprehensive Vector Function Launcher prompt.

The Vector launcher is one of the three modes of the main module launcher, ΣML, by pressing the [A] key at the initial “M:_” prompt – which changes it into the “V:_” prompt.

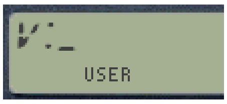
As it occurs with theme modules, there are a large number of functions that work together and ideally should all be available in a USER keyboard for optimal utilization. And as it’s become customary (see the 41Z and SandMath manuals) – a dedicated launch pad is the best solution to solve this need.

With this approach it’s not necessary to make multiple key assignments to quickly access all of the functions – saving memory and allowing for other USER key mappings. The “***V:***” prefix indicates a prompting for the function keys, which are logically mapped to the same real-functions on the standard 41C keyboard. Prompting functions are also included in this implementation.

The picture below shows the Vector Keyboard, as invoked by the **V:** launcher:

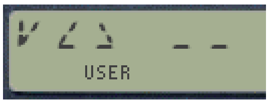


Note that with exception of the conditional tests, all mapped keys are direct, no need to use [SHIFT] unless for INDirect addressing. Note as well that the prompts also support using keys from the two top rows for quick argument entry 1-10



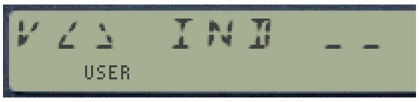
V:_
USER

;

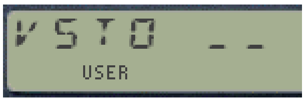


V<> __
USER

;

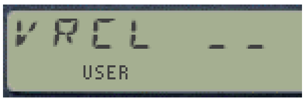


V<> IND __
USER

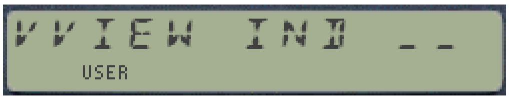


VSTO __
USER

;



VRCL __
USER



VVIEW IND __
USER

Vector Conditionals.

Other useful functions are those used to test whether vectors in V1 and V2 levels are equal/different, or whether the V1 level contains a zero-vector: **V=0?**, **V#0?**, **V=A?**, and **V#A?** They return YES/NO in RUN mode, and skip the next line if false when used in a program.

Note: There is a cosmetic limitation in the subfunction names: even if it's not displayed correctly, the "#" character is [SHIFT] [**H**] in the Alpha keyboard.

A word on Vector Stack implementation.

Before you ask: no; there isn't a memory buffer or any similar fancy trick implemented in the Vectors ROM for a full-fledged vector stack. Because there isn't a dedicated memory area for it, it follows that the implementation must use the standard means for all purposes – meaning the (real) stack and Alpha registers. Even with this limitation it's possible to implement a poor-man's version of a vector-RPN stack with just two levels, plus also featuring "Last-Vector" functionality for the most important cases. The following paragraphs describe the details of such an implementation.

1. **Two stack levels (V1 and V2) using the stack and Alpha.**

   Logically the three components of the first vector level (V1) will occupy the X,Y and Z registers in the stack. *Monadic functions* should return the result in the X register, placing the x-coordinate in LastX for easy retrieval, should the original vector is to be recalled. *Dual functions* will operate on both the stack and the Alpha registers M, N, and O – which hold the second vector operand (V2). In addition to the Vector Recall, Storage and Exchange a few other functions are included for convenient handling of the levels: **VENTER^**, **V<>A**, and **LASTV** (see below). With all these the 2-level vector stack becomes useful and suitable for the majority of usual scenarios.

2. **Fully Programmable VRCL, VSTO, V<>, VVIEW.**

   Using the non-merged functions technique, the numeric argument is taken from the next line in a program. Zero is not explicitly required and INDirect is done by adding 128 to the address (80 Hex). They operate on 3-register blocks, and their indexes are logical and not related to the actual underlying data registers.

3. **Compatibility with Alpha messages:- Scratch.**

   The choice of Alpha for the second vector level (V2) requires some work-around when the use of Alpha prompting or displaying is also needed – in a FOCAL program, like **VC** or **TR**. This has been accomplished with **EXSCR**, a simple register swapping function to be called both before and after the PROMPT or AVIEW actions – effectively preserving V2 in a scratch area: registers **L**,**T**, and "**a**" during the process. This requires that no more than 2 pending subroutine returns are used before calling "***VC***" - not perfect but clever enough to work in practicality.

4. **LASTV functionality:- Level "0" in a dynamic scheme.**

   The natural choices for a temporary storage of the LastV (V0) components are the still-unused registers **L**,**T**, and **P**. Using **L** is a given, since it already works like a *LastX* in the real case. **T** is a safe choice but **P** becomes unusable in all practical cases due to the OS demand for it as data-entry scratch. We chose register **Q** instead, as a better fit despite also being very volatile. In reality both **P** and **Q** are used to hold the third coordinate in sequence: it is first stored in **P** when the function execution commences, where it remains during the calculations – which allows free usage of **Q** for the 13-digit math routines. Then it is transferred from **P** to **Q** upon exiting the function, where it stays for **LASTV** to pickup (until/unless **Q** is used by the OS in other action).

5. **Viewing the vector components.**

   Functions **VIEWV** and **VVIEW** provide a sequential displaying of the three vector components for V1 or any vector stored in memory. The display is used and not Alpha (so level-2 is not disturbed), and each component value is tagged with its own label - X:, Y:, or Z: These functions are also fully programmable. The time lapsed in between components display is fixed, but the displaying will be halted while any key is kept depressed.

New versions of the classic Advantage Programs.

The SandMatrix contains new versions of the popular Vector programs from the Advantage Pac: ***"VC"***, and ***"TR"*** / ***"CT"***. *Their usage is completely compatible*, even if they use all new functions instead of the original FOCAL programs. Obvious advantages are faster execution, shorter program listings, and more accuracy due to the 13-digit math routines employed within the new functions. Also fewer data registers are needed since level-2 is in Alpha's {M,N,O} – and not in {R01, R02, R03} as in the Advantage's case.

Vector operations.

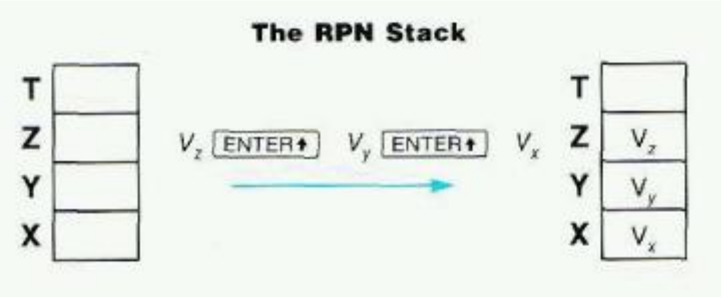
The "**VC**" program simulates a "Vector Calculator" superimposed on your normal calculator. It redefines the functions in the top two rows of keys to these vector operations: addition, subtraction, distance, dot product, cross product, angle between vectors, norm, and unit vector. This pac also offers these operations to you as regular functions (without the Vector Calculator) that you can execute like any other Hp-41 (nonkeyboard) function. Their Alpha names are given under "Summary of Vector Operations*.

The vector operations operate on three-dimensional vectors described in rectangular coordinates. That is, every vector has three components, Vx, Vy, and Vz. For a two-dimensional vector, Vz must be equal to zero.

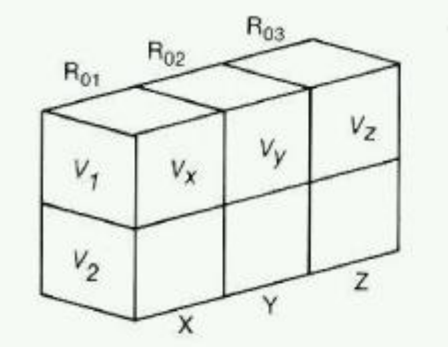
A complement to **VC** is the Coordinate Transformations program, **TR**. This means you can carry out vector operations and transformations on the same data, since you can access either program from the other one. The use of coordinate transformations is covered in the next section, "Coordinate Transformations".

Method.

The Vector Calculator (program VC) creates a vector stack that works in concert with the regular RPN stack (X-, Y-, Z-, and T-registers). When you enter the three components of a vector in the order Vz, Vy, Vx, they occupy the regular stack like so:



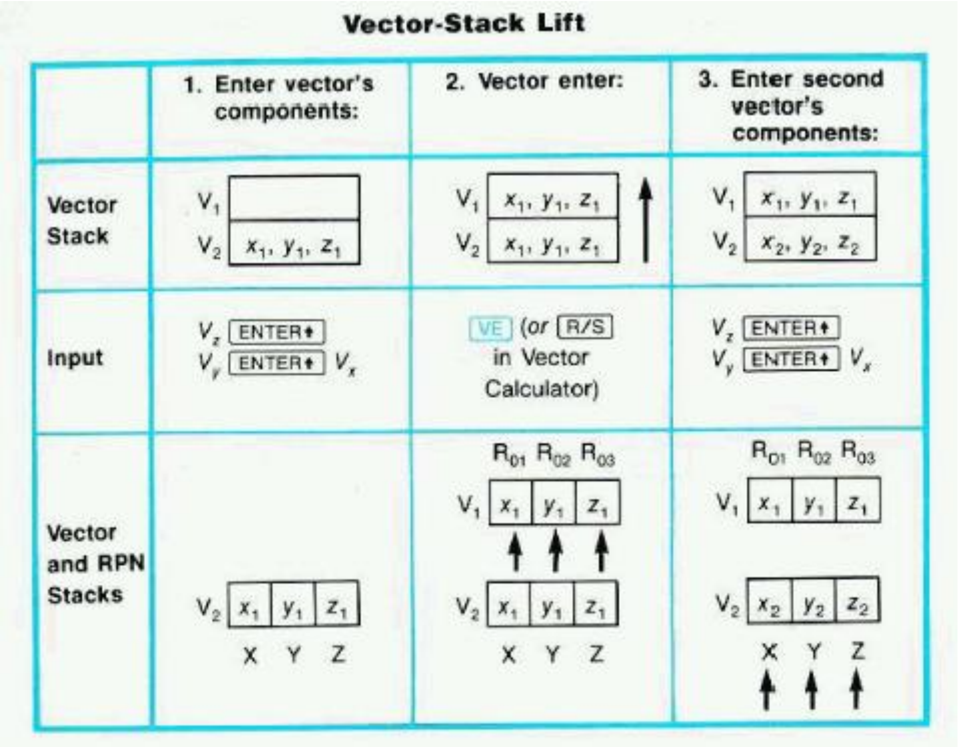
How do the two stacks relate to each other? Basically, the "bottom" level of the vector stack (V2) is stored in registers X, Y, and Z of the stack, while the "upper" level of the vector stack (V1) is stored in ALPHA registers M, N, and O. You can imagine the registers shared in a three-dimensional stack like so:



The Vector stack is two vector-level high, so it accommodates two vectors. Note, however, that each level contaris three components, the x-, y-, z-components far each vector.

The diagram below shows you what happens in vector entry and vector stack movement from the point-of-view of the vector stack and from the point-of-view of the RPN and vector stacks together:

When you enter two vectors (as you would prior to executing a typical vector operation), the first one you key in becomes V1 and the second one you key in becomes V2. A "vector entry" (the function **VENTER^**, or pressing R/S in the Vector Calculator) copies the bottom vector (V2) into the top Vector (V1) . Then, when you key in the next vector, it overwrites the copy in the bottom vector (V2), leaving the first vector in V1 and the second vector in V2.



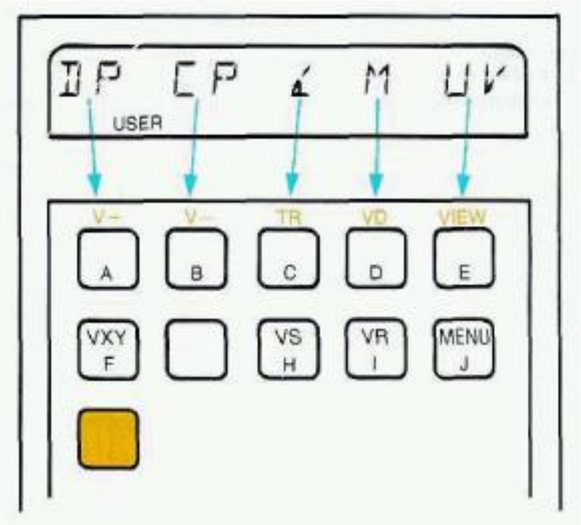
Vector-Stack Lift

	1. Enter vector's components:	2. Vector enter:	3. Enter second vector's components:
Vector Stack	V_1 □ V_2 x_1, y_1, z_1	V_1 x_1, y_1, z_1 ↑ V_2 x_1, y_1, z_1	V_1 x_1, y_1, z_1 V_2 x_2, y_2, z_2
Input	V_z ENTER↑ V_y ENTER↑ V_x	VE (or R/S in Vector Calculator)	V_z ENTER↑ V_y ENTER↑ V_x
Vector and RPN Stacks	V_2 x_1 y_1 z_1 X Y Z	R_{01} R_{02} R_{03} V_1 x_1 y_1 z_1 ↑ ↑ ↑ V_2 x_1 y_1 z_1 X Y Z	R_{01} R_{02} R_{03} V_1 x_1 y_1 z_1 V_2 x_2 y_2 z_2 X Y Z ↑ ↑ ↑

Instructions.

- Starting **VC** (invoking the vector calculator) does not clear the vector stack, so you can still work with previously stored vectors.
- Be sure to give each vector three dimensions. If it has only two dimensions, then enter a zero for Vz.
- Enter the vector's dimensions as rectangular coordinates. If you have polar coordinates (magnitude and angle) for a two-dimensional vector, convert them using the function P-R (polar to rectangular).
- For those operations involving angles, the units will match the current angular mode setting (Degrees, Radians, or Grads).
- The view function ([ ][**E**]) is very useful for reviewing the components of V2 in the stack.
- V1 refers to the "top" vector; the one in {M, N, and O} Alpha registers. V2 refers to the "bottom" vector; the one in {X, Y, and Z} stack registers

This menu will show you which key corresponds to which function in VC. Press [**J**] to recall this menu to the display at any time.



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

The Vector Calculator provides two methods for entering a vector into the vector stack. The vector-enter function (**VENTER^**) is analogous to the ENTER^ key. A shortcut method of vector entry is the R/S key. Whenever you enter the vector components from the keyboard when the menu was the last thing displayed before keying in the three components. Pressing R/S will perform the same function as **VENTER^**.

The following table shows the keystrokes to execute vector operations on the Vector Calculator (program VC). For a definition of each operation, refer to the "Summary of Vector Operations" following the Instruction Table.

Instructions	Key in:	Display
Start the program for the Vector Calculator VC	XEQ "**VC**"	DP CP <) M UV
Enter the three components of your first vector (V1). Separate two vectors with a vector enter after the first set of coordinates: execute **VENTER^** or – if the menu was the last thing displayed before you entered the first component – press R/S.	z1, ENTER^ y1, ENTER^ x1, R/S – or **VENTER^**	z1 y1 DP CP <) M UV
Key in the second vector (V2). Do not press R/S	z2, ENTER^ y2, ENTER^ x2	z2 y2 x2
Display the main menu (optional)	[**J**]	DP CP <) M UV
Execute a vector operation: Dot Product: V1 * V2	 [**A**] (DP)	 DOT = result

Cross Product: V1 x V2	[**B**] (CP)	X= x result
		Y= y result
		Z= z result

Angle between V1 and V2	[**C**] (<)	<) result
Norm (magnitude) of V2	[**D**] (M)	M= result

Unit Vector of V2	[**E**] (UV)	X= x result
		Y= y result
		Z= z result

Vector Add V1+V2	[ ] [**A**]	X= x result
		Y= y result
		Z= z result

Vector Subtract V1-V2
 	
Coordinate Transformations – refer to the next section for instructions.
 	
Distance between V1 and V2
	[ ] [B]
	
	[ ][C]
	USER [ ][C]
	
	[ ][D]
		X= x result
		Y= y result
		Z= z result
		Z0,Y0,X0 ?
		DP CP <) M UV
		
		d= result

Restore the main menu after or between operations (optional)	[J]	DP CP <) M UV

To view the components of V2, the vector in the stack:	[ ][E]	X= x result
		Y= y result
		Z= z result

To exchange V1 and V2 (the vector components in {M,N,O} and {X,Y,Z}	[F]	DP CP <) M UV
To store V2.s components as vector-register "n" in R3n+1, R3n+2, R3n+3:	n, [H]	DP CP <) M UV

To recall the conents of vector-register n into V2 (X,Y,Z), pushing V2 into V1:	n, [I]	X= x result
		Y= y result
		Z= z result

Remarks

You can eliminate the displaying of results on the Vector Calculator by setting flag 04. This lets you perform successive calculations more quickly by not having to step through the display of the results. You can still view the results when you want by pressing [ ][E].

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

Summary of Vector Operations

The vector operations are accessible in two different ways:

- By using the Vector Calculator and its redefined keys, as explained above.
- By directly executing a vector function using its Alpha name, like any other HP-41 nonkeyboard function.
- V1 refers to the first (or "top") vector: the one in M, N and O. V2 refers to the second (or "bottom") vector: the one in X, Y, and Z.

The operations perform the same calculations regardless of how they are executed. These characteristics are given in the table below, along with their Alpha names and descriptions.- You can also execute these operations by Alpha name from inside the Vector Calculator, though it is usually more convenient to use the Vector Calculator's redefined keys .

Table of Vector Functions. This table shows all functions implemented in the module. Note that the convention followed indicates V1 level in {X,Y,Z} and V2 level in {M,N,O} – that is reversed from the Advantage's.

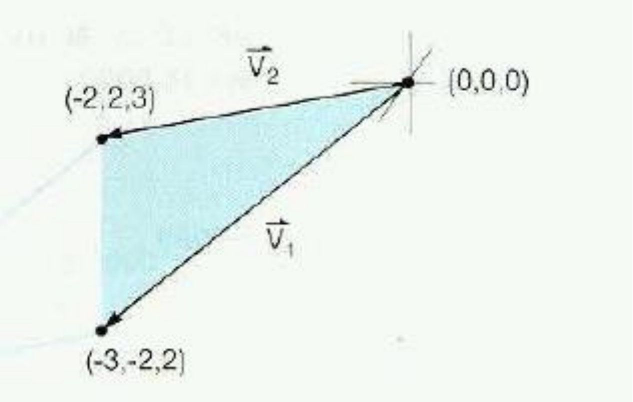
Function	Effect
A+V	Adds vectors V1 and V2. Result vector (v2+v2) is placed in V1, with v1 saved in LastV (V0). V2 is left unchanged.
A-V	Subtracts vectors V1 and V2. Result vector (v2-v1) is placed in V1, with v1 saved in LastV (V0). V2 is left unchanged.
A*V	Calculates the DOT product of V1 and v2. Result is left in X-reg. X-component of v1 is saved in LastX, all other registers are unchanged.
AXV	Calculates the CROSS product of V1 and V2. Result vector (v2 x v1) is placed in V1, with v1 saved in LastV. V2 is left unchanged.
A/V	Calculates V2 x (1/V1). Same result and LastV conventions apply.
EXSCR	Exchanges Vector in Alpha v2 with registers {L,T,a}
LASTV	Recalls last vector to level-1 and pushes V1 into V2
X*V	Scales the vector with components in {T,Z,Y} by the factor in X. Result is left in level V1
V<> _ _	Exchanges vector in V1 with the vector-register nn. Supports INDirect addressing.
V<>A	Exchanges vectors in levels V1 and V2
VADST	Calculates the distance between V1 and V2. Result is left in X-reg, with X-component of v1 saved in LastX.
VANG	Calculates the angle between V1 and V2. Result is left in X-reg, with X-component of v1 saved in LastX.
VCHS	Changes sign of all components of v1. Original v1 is saved in LastV (V0).
VENTER^	Pushes v1 (in X,Y,Z-regs) into the V2 level (M,N,O-regs).
VIEWV	Sequentially shows the three components of vector v1
VINV	Replaces v1 with the inverse of its components. Original v1 is saved in V0.
VMOD	Calculates the modulus of v1 = SQRT(Vx^2+Vy^2+Vz^2). Result is placed in X-reg, with the x component saved in LastX.
VNORM	Calculates the norm of v1 = (Vx^2+Vy^2+Vz^2). Result is placed in X-reg, with the x component saved in LastX.
VRCL _ _	Recalls vector-register nn to V1 level, pushes v1 into V2 – and v2 is lost. Supports INDirect addressing.
VSTO _ _	Stores v1 into vector-register nn. Supports INDirect addressing.
VUNIT	Replaces v1 with its unitary vector, that is v = v1 / |v1|. Original v1 is placed in LastV. V2 is unchanged.

VVIEW _ _	Sequentially views the components of vector-register nn. Supports INDirect addressing.
V=0?	Conditional test on the three components of v1 being zero. Returns YES/NO, skips line in a program if FALSE.
V#0?	Conditional test on at least one component of v1 not zero. Returns YES/NO, skips line in a program if FALSE.
V=A?	Conditional test on v1 being equal to v2. Returns YES/NO, skips line in a program if FALSE.
V#A?	Conditional test on v1 not equal to v2. Returns YES/NO, skips line in a program if FALSE

Remember that of all these, only the prompting functions (with underscores next to the name) are located in the main FAT – and the rest are sub-functions in the secondary FAT, therefore you need to use **ΣV$** or **ΣV$** to execute them.

Example 1.-

Find the area of the triangle determined by the vectors v1 = (- 3, - 2, 2) and v2 = (- 2, 2, 3). Recall that the area of the parallelogram determined by v1 and v2 equals the magnitude of v1 x v2.



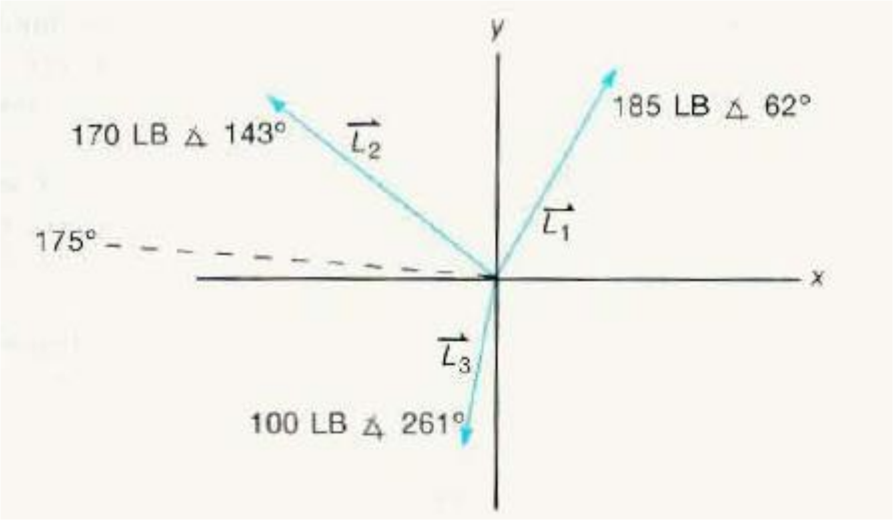
The solution is half the magnitude of the cross product. Using the Vector functions:

Type	Result
3, ENTER^, 2, ENTER^, 2, CHS, **VENTER^**	(-2, 2, 3)
2, ENTER^, 2, CHS, ENTER^, 3, CHS, **AXV**	(10, -5, 10)
VMOD	15,000
2, /	7,500

Example 2.-

Resolve the following three loads along a 175-degree line. Use the dot product on the sum of the three loads to do so. You will first need to convert the polar coordinates to rectangular coordinates. Remember to set z=0.

Save the results for the polar coordinates of L3 and the 175°-line so that you can re-use them to find the resolution (dot product) when L3 is doubled. This example stores those results in vector-registers 1 and 2.



The solution is the dot product of the resulting vector and the unit vector in the desired direction. Each vector must be converted to rectangular coordinates, so here are the keystrokes:

Keystrokes	Action	Result
XEQ “DEG”	Sets angular mode	
0, ENTER^, 143, ENTER^, 170	L1 in polar coordinates	
P-R	rectangular	(-135.768, 102.309, 0)
VENTER^	*pushes it to V2*	
0, ENTER^, 62, ENTER^, 185	L2 in polar coordinates	
P-R	rectangular	(86.852, 163.345, 0)
A+V,	adds L1+L2	(-48.916, 265.654, 0)
VENTER^	*pushes it to V2*	
0, ENTER^, 261, ENTER^, 100	L3 in polar coordinates	
P-R	rectangular	(-15.643, -98.769, 0)
VSTO 01	saves L3	
A+V,	adds L1+L2+L3	(-64.559, 166,885, 0)
VENTER^	pushes it to V2	
0, ENTER^, 175, ENTER^, 1	line polar coordinates	
P-R	rectangular	(-0.996, 0.087, 0)
VSTO 02	saves unit vector	
A*V	calculates projection	78.859

Note that you need to execute **VENTER^** at the end of each intermediate vector calculation, so the result is placed in the V2 level (in registers M,N,O) and doesn’t get overwritten by the new vector components being entered.

When the load L3 is doubled, since the sum [L1+L2+L3] is still in the level-2, we type:

Keys	Action	Result
V<>A	brings L1+L2+L3 back to level-1	
VRCL 01	pushes V1 into V2 and recalls L3 to V1	
A+V	adds L3 to the previous sum	
VRCL 02	pushes V1 into V2 and recalls the line unit vector	
A*V	calculates the projection	**85,834**

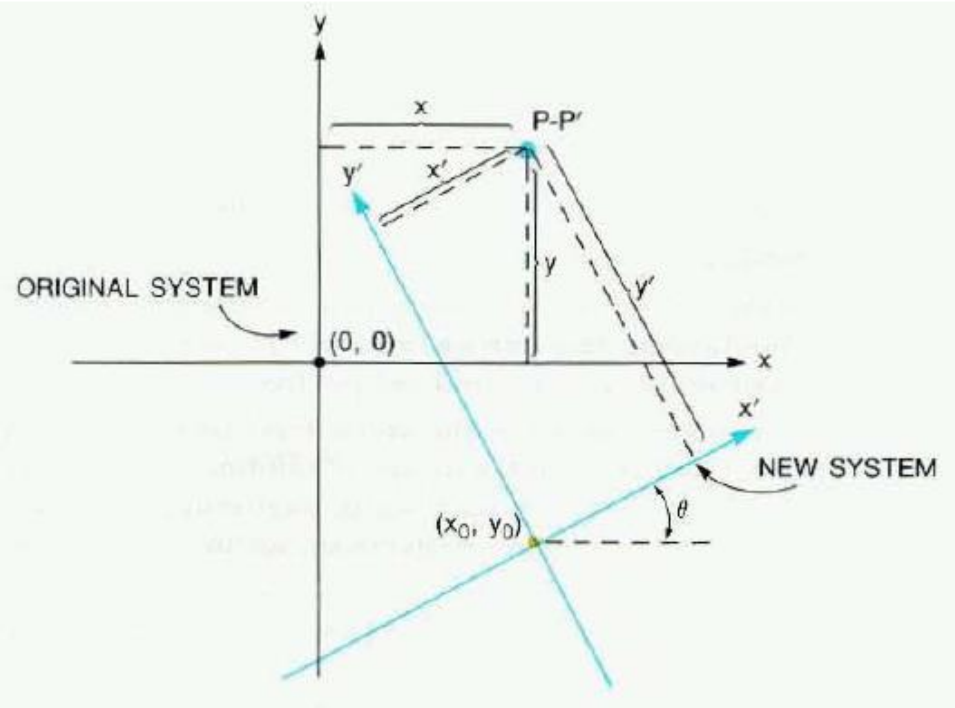
There’s nothing surprising about the utilization of the vector functions, which use the same logic to separate vector arguments as the complex numbers in the 41Z module (using ZENTER^); that you may already be familiar with.

Coordinate transformations

The **TR** program performs three-dimensional translation of coordinates, with or without rotation. This program uses parts of the **VC** program for vector operations. You can access **TR** either directly or from VC.

The program prompts you for the coordinates of the origin of the new system (x0, y0, z0), the angle of rotation of this system relative to the original system, and the axis about which the rotation is performed. You can then enter points in the original system (x, y, z) that you want transformed to the new system (x', y', z'), or enter points in the new system (x',y',z') that you want transformed to the original system (x, y, z). For a two-dimensional case, enter z0 as zero.

A Two-Dimensional Rotation about the Axis (0, 0, 1)



After specifying the new origin (x, y, z), you specify the rotation angle. For a three-dimensional system with a non-Zero angle of rotation, you also specify its rotation vector (a, b, c). The rotation vector defines the axis about which the rotation is to be done; it can have any non-zero magnitude. Two-dimensional transformations are handled as a special case of three-dimensional transformations with (a, b, c) set to (0, 0, 1).

Equations

$$\vec{P'} = [(\vec{P} - \vec{T}) \cdot \vec{n}]\,\vec{n}\,(1 - \cos\theta) + (\vec{P} - \vec{T})\cos\theta + [(\vec{P} - \vec{T}) \times \vec{n}]\sin\theta$$

$$\vec{P} = [(\vec{P'} \cdot \vec{n})\,\vec{n}\,(1 - \cos\theta) + \vec{P'}\cos\theta + (\vec{P'} \times \vec{n})\sin(-\theta)] + \vec{T}$$

where:

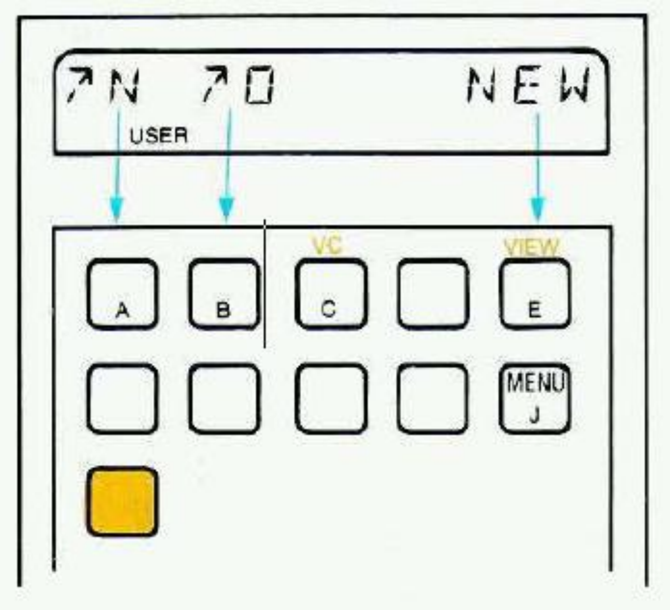
$\vec{P'}$ = new system coordinates
$\vec{P}$ = old system coordinates
$\vec{T}$ = origin of new system
$\vec{n}$ = unit rotation vector (a, b, c)
θ = rotation angle

Instructions

You can start **TR** either directly (XEQ "TR") or from the Vector Calculator ([ ][**C**]) in **VC**. The Vector Calculator is covered in the "Vector Operations" chapter.

Enter coordinates as rectangular coordinates and specify angles according to the current setting (Degrees, Radians, or Grads mode).

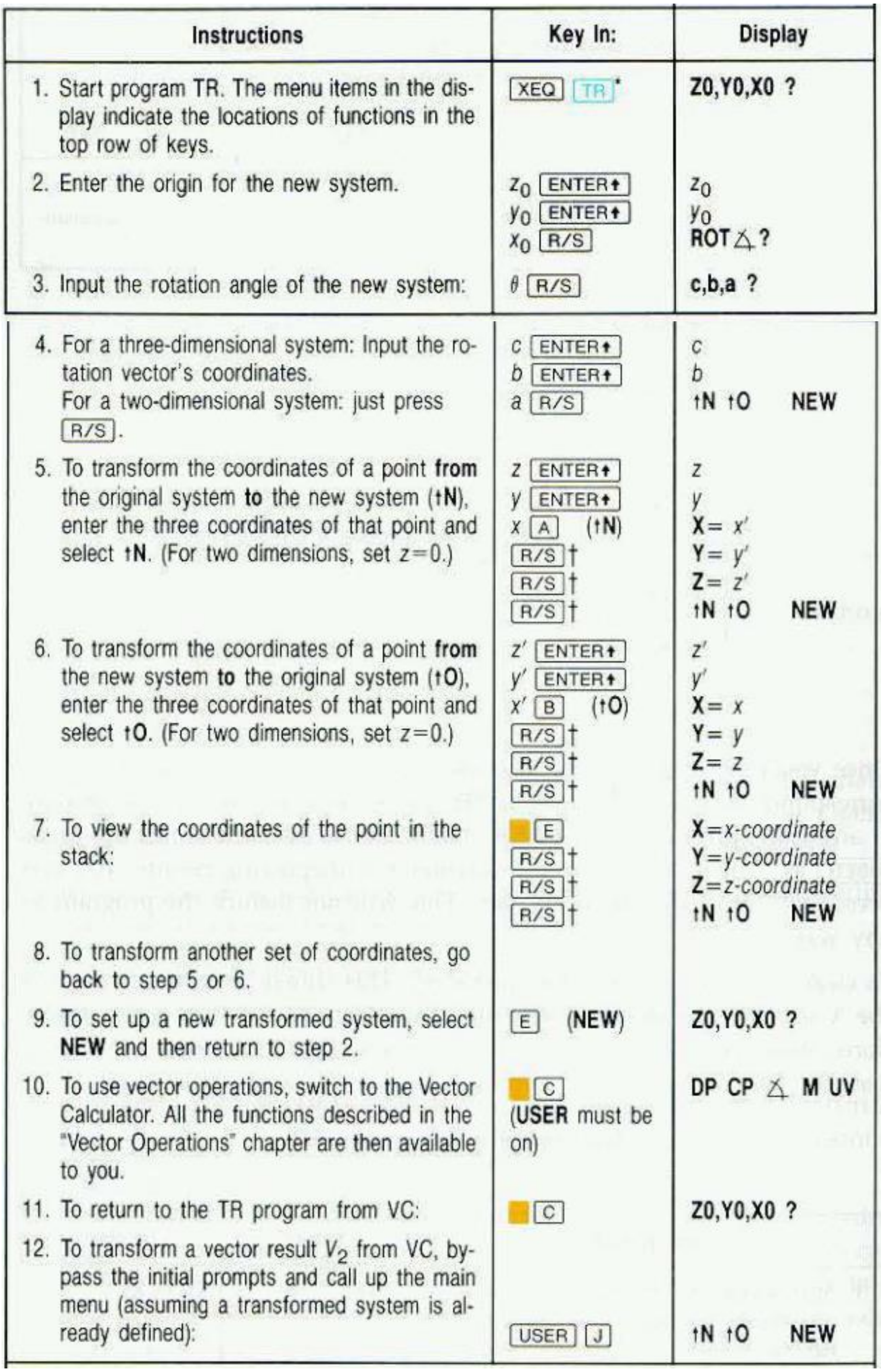
- For two dimensions, input zero for the z-value.
- For pure translation, input zero for the rotation angle.
- For pure rotation, input zeros for x0, Y0, and z0
- The sign of the rotation angle is determined by the right-hand rule and the direction of the rotation vector. For two dimensions, counter-clockwise rotation is considered positive.
- You can switch into and out of the Vector Calculator by pressing . ([ ][**C**] ("C" for Calculator and Coordinate transformations). You can then perform vector operations upon vector coordinates in the stack and in storage registers. (Refer to "Remarks' for the storage locations of the vector coordinates.)
- The view function ([ ][**E**]) is very useful for reviewing the coordinates of the point in the stack.



Once you have entered your variables, this menu shows you which key corresponds to which function in **TR**. To restore this menu to the display at any time, press [**J**] *if the USER annunciator is On*. (If it is not on, press USER to turn it on.) Or, if the calculator is displaying results, you can press R/S until the menu appears. This will not disturb the program in any way.

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

Instruction Table for TR



Instructions	Key In:	Display
1. Start program TR. The menu items in the display indicate the locations of functions in the top row of keys.	XEQ TR*	Z0,Y0,X0 ?
2. Enter the origin for the new system.	z_0 ENTER↑ y_0 ENTER↑ x_0 R/S	z_0 y_0 ROT△?
3. Input the rotation angle of the new system:	θ R/S	c,b,a ?
4. For a three-dimensional system: Input the rotation vector's coordinates. For a two-dimensional system: just press R/S.	*c* ENTER↑ *b* ENTER↑ *a* R/S	*c* *b* ↑N ↑O NEW

5. To transform the coordinates of a point **from** the original system **to** the new system (↑**N**), enter the three coordinates of that point and select ↑**N**. (For two dimensions, set $z=0$.)	*z* ENTER↑
	y ENTER↑
	x A (↑N)
	R/S †
	R/S †
	R/S †
		z
		y
		X= x'
		Y= y'
		Z= z'
		↑N ↑O NEW

6. To transform the coordinates of a point **from** the new system **to** the original system (↑**O**), enter the three coordinates of that point and select ↑**O**. (For two dimensions, set $z=0$.)	z' ENTER↑
	y' ENTER↑
	x' B (↑O)
	R/S †
	R/S †
	R/S †
		z'
		y'
		X= *x*
		Y= *y*
		Z= *z*
		↑N ↑O NEW

7. To view the coordinates of the point in the stack:	■ E
	R/S †
	R/S †
	R/S †
		X=*x-coordinate*
		Y=*y-coordinate*
		Z=*z-coordinate*
		↑N ↑O NEW

8. To transform another set of coordinates, go back to step 5 or 6.		
9. To set up a new transformed system, select **NEW** and then return to step 2.	E (**NEW**)	Z0,Y0,X0 ?
10. To use vector operations, switch to the Vector Calculator. All the functions described in the "Vector Operations" chapter are then available to you.	■ C (**USER** must be on)	DP CP △ M UV
11. To return to the TR program from VC:	■ C	Z0,Y0,X0 ?
12. To transform a vector result V_2 from VC, bypass the initial prompts and call up the main menu (assuming a transformed system is already defined):	USER J	↑N ↑O NEW

Remarks

This program uses local Alpha labels (as explained in the owner's manual for the HP-41) assigned to keys [**A**], [**B**], [**E**], [ ][**C**]], and [**J**]. These local assignments are *overridden* by any User-key assignments you might have made to these same keys, thereby defeating this program. Therefore be sure to *clear any existing User-key assignments of these keys* before using this program, and avoid redefining these keys in the future.

However, these local Alpha labels are active only while the USER annunciator is on. This allows you to use the arithmetic functions in the top two rows while the USER annunciator is off. (As long as USER is on, the keys mentioned above are redefined and will not execute their Normal functions.)

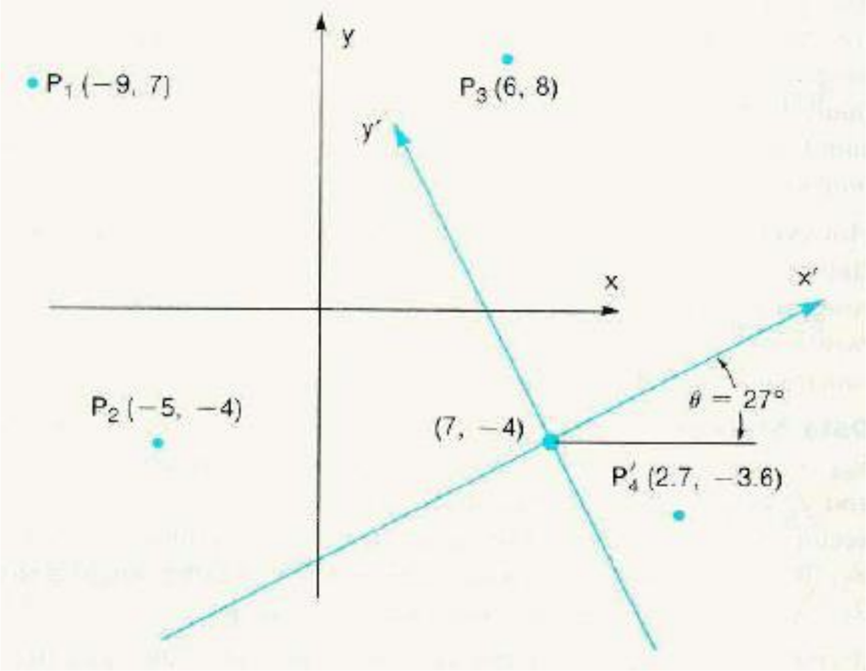
Data Storage. The vector or point you want to transform is stored in R00, R01, R02, which is vector-storage register 0 (initially from the X-, Y-, and Z-registers). The rotation vector is stored in R03, R04, R05, which is vector-storage register 1. The origin of the new system is stored in R06, R07, R08, which is vector-storage register 2. The rotation angle is stored in R13, while R09 , R10, and R11 are used for scratch.

If you will be using vector storage operations (**VSTO**, **VRCL**, and the Vector Calculator) along with **TR**, keep in mind that **TR** uses R0-R13 when it is initialized (XEQ "TR"). This means you should not store vectors in vector registers 1 through 4 (if you plan to use **TR** in your vector calculations).

Flags. Flag 01 is used to indicate whether the transformation is to be made to the new system or to the original system. When flag 1 is set, the transformation is to the new system. Flag 05 is set when the system is rotated.

Example 1.-

The coordinate systems (x, y) and (x', y') are shown below. Convert the points P1,' P2, and P3 to equivalent coordinates in the (x', y') system. Convert the point P’4 to equivalent coordinates in the (x, y) system.



Keystrokes	Display	Action
FIX 4		sets the display format used here.
XEQ " TR"	"Z0,Y0,X0?"	prompts for new origin
0, ENTER^, 4, CHS, ENTER^, 7	7.0000	components in stack
R/S	"ROT<) ?"	prompts for angle of rotation
27, R/S	"c,b,a ?"	prompts for the rotation vector
R/S (2-dimensional)	"7N 70 NEW"	ready for P1
0, ENTER^, 7, ENTER, 9, CHS,	-9.0000	components in stack

[A] (^N)	X=-9.2622
	Y= 17.0649
	Z= 0.0000
		shows new components
		(in automated sequence)

R/S (optional)	"7N 70 NEW"	ready for P2
0, ENTER^, 4, CHS, ENTER^, 5, CHS	-5.0000	components in stack

[A] (^N)	X=-10.5921
	Y=5.4479
	Z=0.0000
		shows new components
		(in automated sequence)

R/S (optional)	"7N 70 NEW"	ready for P3
0, ENTER^, 3.6, CHS, ENTER^, 2.7	2.7000	components in stack

[B] (^O)	X= 11.0401
	Y=-5.9818
	Z=0.0000
		shows old components
		(in automated sequence)

Example 2.-

A three-dimensional coordinate system is translated to (2.45, 4.00, 4.25). After the translation, a 62.5 degree rotation occurs about the (0, - 1, - 1) axis. In the original system, a point had the coordinates (3.9, 2.1, 7.0). What are the coordinates of the point in the translated, rotated sytem?

Keystrokes	Display	Action
[J]	· 7N 70 NEW"	retrieves menu (if USER is on)
[E] (NEW)	"Z0,Y0,X0?"	prompts for new origin
4.25, ENTER^, 4, ENTER, 2.45	2.4500	components in stack
R/S	"ROT<) ?"	prompts for angle of rotation
62.5, R/S	"c,b,a ?"	prompts for the rotation vector
1, CHS, ENTER^, ENTER^, 0	0.0000	components in stack
R/S	"7N 70 NEW"	ready for P
7, ENTER^, 2.1, ENTER^, 3.9	3.90000	components in stack

[A] (^N)	X=3.5861
	Y=0.2609
	Z=0.5891
		shows old components
		(in automated sequence)

Programming Information

The subroutine **CT** can be used in your own programs. It performs coordinate transformations (rotations and translations) in three dimensions. It takes the x- , y-, and z-values from the stack (X-, Y-, and Z-registers) and transforms them to another system, or from the new system to the original system.

Comments. To use **CT**, load the translation vector (T), the unit rotation vector (N), and the rotation angle, set flag 01 to go to the new system or clear flag 01 to go to the original system. Set flag 05 to rotate the vector's coordinates (P). The result is returned to the X-, Y-, and Z-registers and in R1, R02 and R03.

Note that **CT** is located in the secondary FAT, thus it requires **ΣV$** (or **ΣV#**) to run.

Program listings

As you can see by comparing it to the original programs the code length is drastically reduced – and the program clarity is also much improved, with a very intuitive structure behind all functionality.

```
01  LBL "TR"
02  SIZE?
03  14
04  X>Y?
05  PSIZE
06  LBL E
07  CF 27
08  "Z0,Y0,X0 ?"
09  PROMPT
10  VSTO
11  2
12  CLX
13  "ROT<)?"
14  PROMPT
15  CF 05
16  X=0?
17  GTO 00
18  SF 05
19  STO 13
20  CLST
21  E
22  X<> Z
23  "c,b,a ?"
24  PROMPT
25  VUNIT
26  VSTO
27  1
28  LBL J
29  LBL 00
30  SF 27
31  SF 21
32  CF 01
33  "^N ^O   NEW"
34  PROMPT
35  GTO 00
36  LBL c
37  XROM "3VC"
38  LBL A
39  SF 01
40  GTO 03
41  LBL B
42  CF 01
43  LBL 03
44  XROM "CT"
45  LBL e
46  VIEWV
47  GTO 00
48  LBL "CT"
49  FC? 01
50  GTO 02
51  VRCL
52  2
53  A-V
54  FC? 05
55  RTN
56  LBL 02
57  FC? 05
58  GTO 01
59  VSTO
60  VRCL
61  1
62  AXV
63  RCL 13
64  FC? 01
65  CHS
66  SIN
67  X*V
68  E
69  STO 12
70  RDN
71  VRCL
72  RCL 13
73  COS
74  ST- 12
75  X*V
76  A+V
77  VSTO
78  3
79  VRCL
80  1
81  VRCL
82  A*V
83  ST* 12
84  V<>A
85  RCL 12
86  X*V
87  VRCL
88  3
89  A+V
90  FS? 01
91  RTN
92  LBL 01
93  VRCL
94  2
95  A+V
96  END
```

```
01  LBL "3VC"
02  CF 04
03  LBL J
04  CF 21
05  SF 27
06  EXSCR
07  "DP CP a M UV"
08  AVIEW
09  EXSCR
10  STOP
11  VENTER^
12  GTO J
13  LBL d
14  VADST
15  EXSCR
16  "|V-A|"
17  LBL 02
18  "="
19  ARCL X
20  AVIEW
21  EXSCR
22  STOP
23  GTO J
24  LBL c
25  XROM "TR"
26  LBL C
27  VANG
28  EXSCR
29  "VA<)"
30  GTO 02
31  LBL F
32  V<>A
33  GTO J
34  LBL a
35  A+V
36  GTO 05
37  LBL b
38  A-V
39  LBL 05
40  FS? 04
41  GTO J
42  LBL e
43  VIEWV
44  STOP
45  GTO J
46  LBL B
47  AXV
48  GTO 05
49  LBL H
50  VSTO
51  GTO J
52  LBL I
53  VRCL
54  GTO 05
55  LBL A
56  A*V
57  EXSCR
58  "V*A"
59  GTO 02
60  LBL E
61  VUNIT
62  GTO 05
63  LBL D
64  VMOD
65  EXSCR
66  "|V|"
67  GTO 02
68  END
```

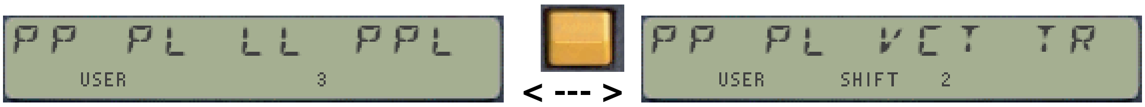
Calculating 2D and 3D Distances: The ΣDST Launcher

A few example programs in the module illustrate the convenience of using an abstraction layer to handle vector expressions – without having to worry about their individual components. The net result is shorter, neater programs with higher-level structure, devoid of all those multiple STOnn / RCLnn statements so common otherwise – and therefore much easier to understand and debug.

The functions included in this group are as follows:

Elements	**3D**	**2D**	**N-Dim**
Point-Point	**VADST**	**PP2**	**DOTN**
Point-Line	**“PL3”**	**PL2**	**-**
Point-Plane	**“PPL3”**	**-**	**-**
Line-Line	**“LL3”**	**-**	**-**

For further convenience these functions are grouped together in its own launcher, **ΣDST** – which is accessed either by pressing XEQ at the “V:” or “P:” prompts; or by pressing USER at any of the three prompts. Once invoked, you can toggle the 2D and 3D cases with the [SHIFT] key, whereby changing the displayed choices accordingly. Note also the usage of the user flags annunciators 2/3 to denote the corresponding case (where the Shifted screen is for the 2D case).



Note that **VCT** and **TR** are shortcuts for the Advantage’s Vector Calculator and Coordinate Transformation programs described before. The formulas used are as follows.-

2D Point-to-Point distance:

$$\mathrm{d}(\mathbf{p},\mathbf{q}) = \sqrt{(p_1-q_1)^2 + (p_2-q_2)^2}.$$

Where the points are given by P(p1,p2) and Q(q1,q2).
Enter q2, q1, p2, p1 in the stack; then execute **PP2.** The result is left in the X-register.

Example: The distance between points M(1,2) and N(3,4) is:

4, ENTER^, 3, ENTER^, 2, ENTER^, 1, **ΣV$** “PP2” -> d= 2,828427125

2D Point-to-Line distance:

$$d = \frac{|mx_1 - y_1 + k|}{\sqrt{m^2+1}}$$

where the line is given by the equation Y= mx+k, and the point is P(x1,y1).

Enter m,p,y1,x1 in the stack; then execute **PL2**. The result is left in the X-register.

Example: The distance from the point M(2,5) and the line y= 3x + 4 is:

3, ENTER^, 4, ENTER^, 5, ENTER^, 2, **ΣV$** “PL2” --> d= 1.581138830

3D Point-to-Point distance. $$d(p,q) = \sqrt{(p_1 - q_1)^2 + (p_2 - q_2)^2 + (p_3 - q_3)^2}.$$

Where the points components are P(p1, p2, p3) and Q(q1, q2 ,q3).
Use **VENTER^** to separate v1 and v2 in the vector stack; then use **VADST**.

Example: verify that the distance between points P(1,2,3) and Q(-1,-2,-3) is the double of the magnitude of any of them.

```
3, ENTER^, 2, ENTER^, 1, VENTER^, VCHS, VADST  -->   d=7,483314774
3, ENTER^, 2, ENTER^, 1, VMOD, 2, *            -->   7,483314774
```

3D point-to-line distance: d = || UxAM || / || U ||

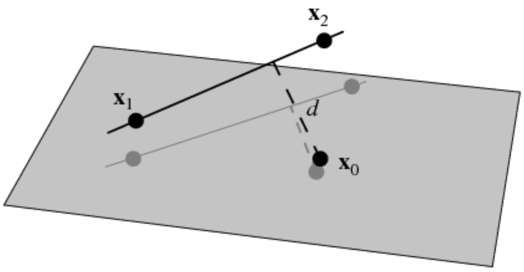
Where the point is M(x,y,z) and the line is determined by its anchor point A(a,b,c) and unit vector U(u,v,w). The program **PL3** will prompt for the components of A, M, and U.

Example: L: is defined by the point A(2,3,4) & the vector U(1,4,9) and M(2,5,3)

```
XEQ " PL3"                  "↗M=?"
3, ENTER^, 5, ENTER^, 2, R/S   "↗A=?"
4, ENTER^, 3, ENTER^, 2, R/S   "↗U=?"
9, ENTER^, 4, ENTER^, 1, R/S   d=2,233785110
```

3D Point-to-Plane distance. For a plane given by (ax + by + cz + d) = 0, and a point (x1, y1, z1) not necessarily lying on the plane, the shortest distance from to the plane is:

$$D = \frac{|ax_1 + by_1 + cz_1 + d|}{\sqrt{a^2 + b^2 + c^2}}$$



The program **PPL3** will prompt for the components, leaving the result in the X-register.

Example: P: 2x + 3y + 5z = 9 and M(4,6,1)

```
XEQ "PPL3"                              "↗PL=?"
9, ENTER^, 5, ENTER^, 3, ENTER^, 2, R/S   "↗M=?"
1, ENTER^, 6, ENTER^, 4, R/S              d=3,568871265
```

3D Line-to-Line distance. d = | (UxU').AA' | / || UxU' ||

For a line L determined by one point A(a,b,c) and one direction vector U(u,v,w) , and another line L' determined by its anchor point A'(a',b',c') and one direction vector U'(u',v',w'). The program **LL3** will prompt for the different components, leaving the result in the X-register.

Example: (L) is defined by A(2,3,4) & U(1,4,7), and (L') is defined by A'(2,1,6) & U'(2,9,5)

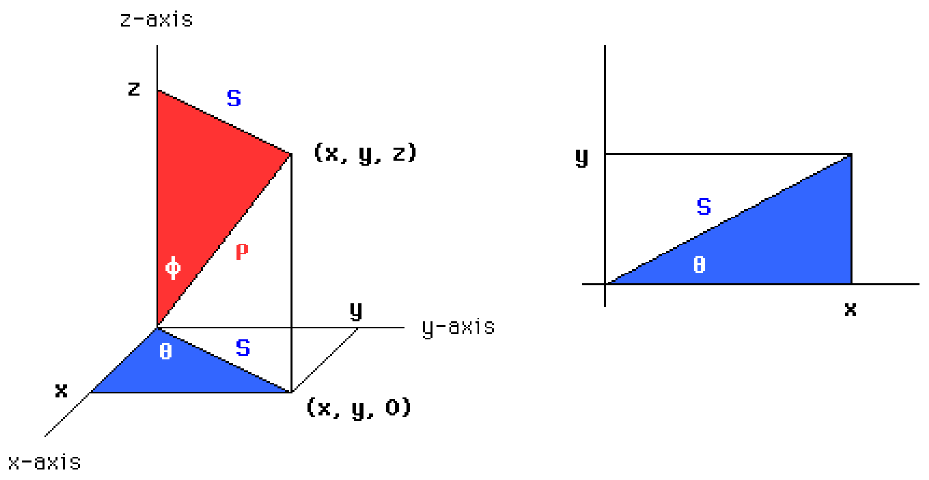
```
XEQ "LL3"                     "↗A'=?"
6, ENTER^, 1, ENTER^, 2, R/S   "↗A=?"
4, ENTER^, 3, ENTER^, 2, R/S   "↗U'=?"
5, ENTER^, 9, ENTER^, 2, R/S   "↗U=?"
7, ENTER^, 4, ENTER^, 1, R/S   d=0,364106847
```

The programs are shown below – really simple when the right functions are used.

```
01  LBL "PL3"
02  "^M=?"
03  XEQ 01
04  XEQ 02
05  VUNIT
06  AXV
07  VMOD
08  GTO 05
09  LBL 01
10  PROMPT
11  VENTER^
12  EXSCR
13  "^A=?"
14  PROMPT
15  EXSCR
16  A-V
17  RTN
18  LBL 02
19  VENTER^
20  EXSCR
21  "U^=?"
22  PROMPT
23  EXSCR
24  RTN
25  LBL "LL3"
26  XEQ 01
27  VSTO (00)
28  "^U'=?"
29  PROMPT
30  XEQ 02
31  AXV
32  VUNIT
33  VRCL (00)
34  A*V
35  GTO 05
36  LBL "PPL3"
37  "^PL=?"
38  PROMPT
39  R^
40  STO 00
41  RDN
42  VENTER^
43  EXSCR
44  "^M=?"
45  PROMPT
46  EXSCR
47  A*V
48  ST- 00
49  V<>A
50  VMOD
51  ST/ 00
52  X<> 00
53  LBL 05
54  ABS
55  "d="
56  ARCL X
57  AVIEW
58  END
```

Coordinate Conversions – SandMath Module.

Functions **R-S** and its inverse **S-R** will convert rectangular to spherical coordinates and back. The convention used is shown in the figure below, with the azimuth angle (theta) measured in the XY plane between the X-axis and the projection of the vector S -, and the Zenith angle (phi) measured from the Z-axis to the vector the radius P. The calculations are made using the internal [TOPOL] and [TOREC] OS routines, same ones used for the native **P-R** and **R-P** functions.



N-dimensional Vector Operations { DOTN , V*VN }

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 values 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 lower 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 inner 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 (if not already there) we execute the function, which returns:

ΣV$ "**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	

This is 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.

Note: The SandMath module includes functions **IN** and **OUT** in the auxiliary FAT (therefore you'd need to use **ΣF$** to execute them) that can be used for data entry in the n-dimensional case, as follows:

IN / **OUT**, sequentially enter data or review a block of registers:

- Enter the initial register index for **IN,** then proceed with all required entries and terminate with a "blank" R/S to end the sequence.
- Input the control word in X in the form bbb.eee, and **OUT** will display all registers sequentially. Use flag 21 to control the display prompt (set) or not (clear).

Initially keeping track of the different sub-function launchers can be a bit challenging , but easy enough to remember that it's just two of them: **ΣF$** in the SandMath and **ΣV$** in the SandMatrix (or their corresponding index-based counterparts **ΣF#** and **ΣV#**)

The alternative – Vectors as Matrices.

V*VN 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 result, multiplying the first operand vector by the transpose of the second operand vector. All data input/output are driven under program control. The execution time is longer than **DOTN**, trading so convenience for speed.

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

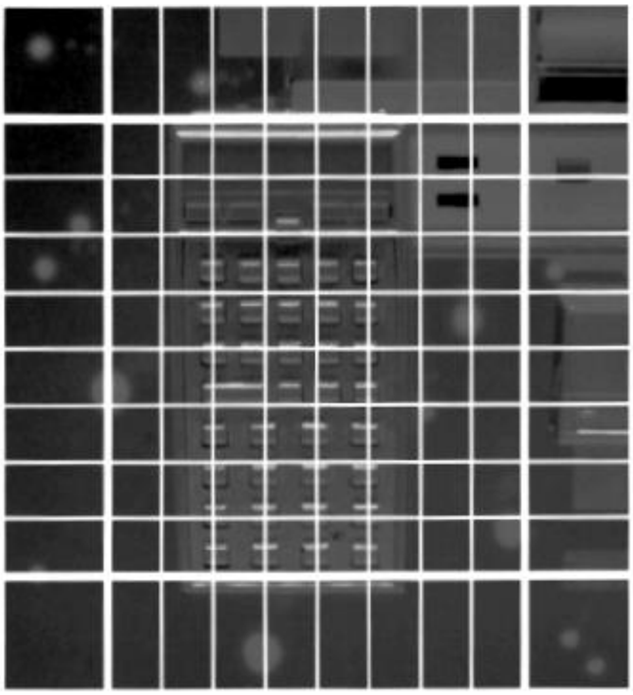
Line 4 uses the header function "-SNDMTRX 4", which in program mode adds the text "ORDER=?" to the display (not ALPHA). This saves bytes and keeps the contents of ALPHA unchanged.

Note: - Functions **EV3** and **PF>X** are handy companions to the main programs **EV3X3** and **PRMF** respectively**.** They've been placed in this section of the auxiliary FAT purely on a convenience basis, to make room in the main FAT for other more relevant functions – so they're available using the sub-function launchers **ΣVL$** or **ΣVL#**. Refer to the corresponding sections in the manual for usage instructions and details.

Remember to always check Jean-Marc Baillard's pages on these and related subjects, really a treasure trove of solid programs you can use. In particular, the Euclidean distances and the Vector products:

http://hp41programs.yolasite.com/distance.php
http://hp41programs.yolasite.com/dotcross.php

Appendices.



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

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]

67	ALIST		450F	161	?NC XQ	Clear LCD and reset things
68	ALIST		4510	124	->4958	[EXIT3]
69	ALIST		4511	175	?NC XQ	Adjust F10 Status
70	ALIST		4512	114	->455D	[ADJF10]
71	ALIST		4513	31D	?NC GO	Normal Function ReturnKB
72	ALIST		4514	002	->00C7	[NFRKB]
73	ALIST	**DELCHR**	**4515**	**3B8**	**READ 14(d)**	to delete rightmost chr
74	ALIST		4516	158	M=C ALL	save it for later
75	ALIST		4517	149	?NC XQ	Disable PER, enable RAM
76	ALIST		4518	024	->0952	[ENCP00]
77	ALIST		4519	178	READ 5(M)	
78	ALIST		451A	2EE	?C#0 ALL	anything in Alpha?
79	ALIST		451B	***037***	***JC +06***	yes, go on
80	ALIST		451C	104	CLRF 8	no, abort if empty
81	ALIST		451D	1B1	?NC XQ	Mainframe Message
82	ALIST		451E	070	->1C6C	[MSGA]
83	ALIST		451F	03C	"NULL"	from table
84	ALIST	fixed bug	**4520**	***37B***	***JNC -17d***	Reset everything and leave
85	ALIST		4521	2E5	?NC XQ	remove last Alpha char
86	ALIST		4522	110	->44B9	[ABSP4]
87	ALIST		4523	198	C=M ALL	recall deleted char value
88	ALIST		4524	106	A=C S&X	store in A for comparisons
89	ALIST		4525	130	LDI S&X	check for SPACE
90	ALIST		4526	020	***"space"***	<space>
91	ALIST		4527	0AD	?NC XQ	complete the logic
92	ALIST		4528	114	->452B	[CHUNK4]
93	ALIST		4529	381	?NC GO	repeat the prompt
94	ALIST		452A	112	->44E0	[GOBACK]
95	ALIST	**CHUNK4**	**452B**	366	?A#C S&X	carry if different
96	ALIST		452C	01F	JC + 03	
97	ALIST		452D	348	SETF 12	allow new space entry
98	ALIST		452E	***0A3***	***JNC +20d***	BAIL OUT
99	ALIST		452F	130	LDI S&X	check for "-" chr
100	ALIST		4530	02D	***"-"***	"-" char value
101			4531	366	?A#C S&X	carry if not "-"
102	Executed within [DELCHR]		4532	02F	JC + 05	
103	an opportunistic routine		4533	***34C***	***?FSET 12***	is there SPACE chr?
104	just grouping common code		4534	***017***	***JC +02***	
105			4535	0C4	CLRF 10	allow new "-" entry
106	ALIST		4536	***063***	***JNC +12d***	BAIL OUT
107	ALIST		4537	198	C=M ALL	recall deleted char value
108	ALIST		4538	3D8	C<>ST XP	Got a radix? If so, we neet to
109	ALIST		4539	14C	?FSET 6	replace it without comma
110	ALIST		453A	***043***	***JNC +08***	
111	ALIST		453B	3D9	?NC XQ	Enable Display (not cleared)
112	ALIST		453C	01C	->07F6	[ENLCD]
113	ALIST		453D	144	CLRF 6	remove the radix value
114	ALIST		453E	284	CLRF 7	(both if need be)
115	ALIST		453F	3D8	C<>ST XP	recall deleted char value
116	ALIST		4540	***3E8***	***WRIT 15(e)***	write it in display
117	ALIST		4541	***184***	***CLRF 11***	Re-allow comma writing
118	ALIST		4542	**3E0**	**RTN**	
119	ALIST	**RADIX4**	**4543**	149	?NC XQ	Disable PER, enable RAM
120	ALIST		4544	024	->0952	[ENCP00]
121	ALIST		4545	3B8	READ 14(d)	put F28 to F9
122			4546	2BC	RCR 7	
123	transfer staus of UF28 to F9,		4547	**248**	**SETF 9**	
124	adds the converted crh code		4548	1EE	C=C+C ALL	comma or period ?
125	to the LCD and prepares ALPHA		4549	013	JNC +02	overflows if COMMA (cf28)
126			454A	**244**	**CLRF 9**	comma = CF 28
127	ALIST		454B	3D9	?NC XQ	Enable Display (not cleared)
128	ALIST		454C	01C	->07F6	[ENLCD]
129	ALIST		454D	**3B8**	**READ 14(d)**	read right
130	ALIST		454E	3D8	C<>ST XP	
131	ALIST		454F	148	SETF 6	
132	ALIST		4550	**24C**	**?FSET 9**	comma or period ?
133	ALIST		4551	013	JNC +02	
134	ALIST		4552	288	SETF 7	should replace the last chr
135	ALIST		4553	3D8	C<>ST XP	with the same one w/ radix
136	ALIST		4554	***3E8***	***WRIT 15(e)***	9-bit LCD write
137	ALIST		4555	130	LDI S&X	
138	ALIST		4556	02C	***","***	appends "," [02C]
139	ALIST		4557	**24C**	**?FSET 9**	
140	ALIST		4558	***360***	***?C RTN***	no need, return
141	ALIST		4559	226	C=C+1 S&X	
142	ALIST		455A	226	C=C+1 S&X	appends "." [02E]
143	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
- Extended-memory support for Polynomial files
- Anything else you feel like going for!

```
.END. REG 163
USER      PRGM
```

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