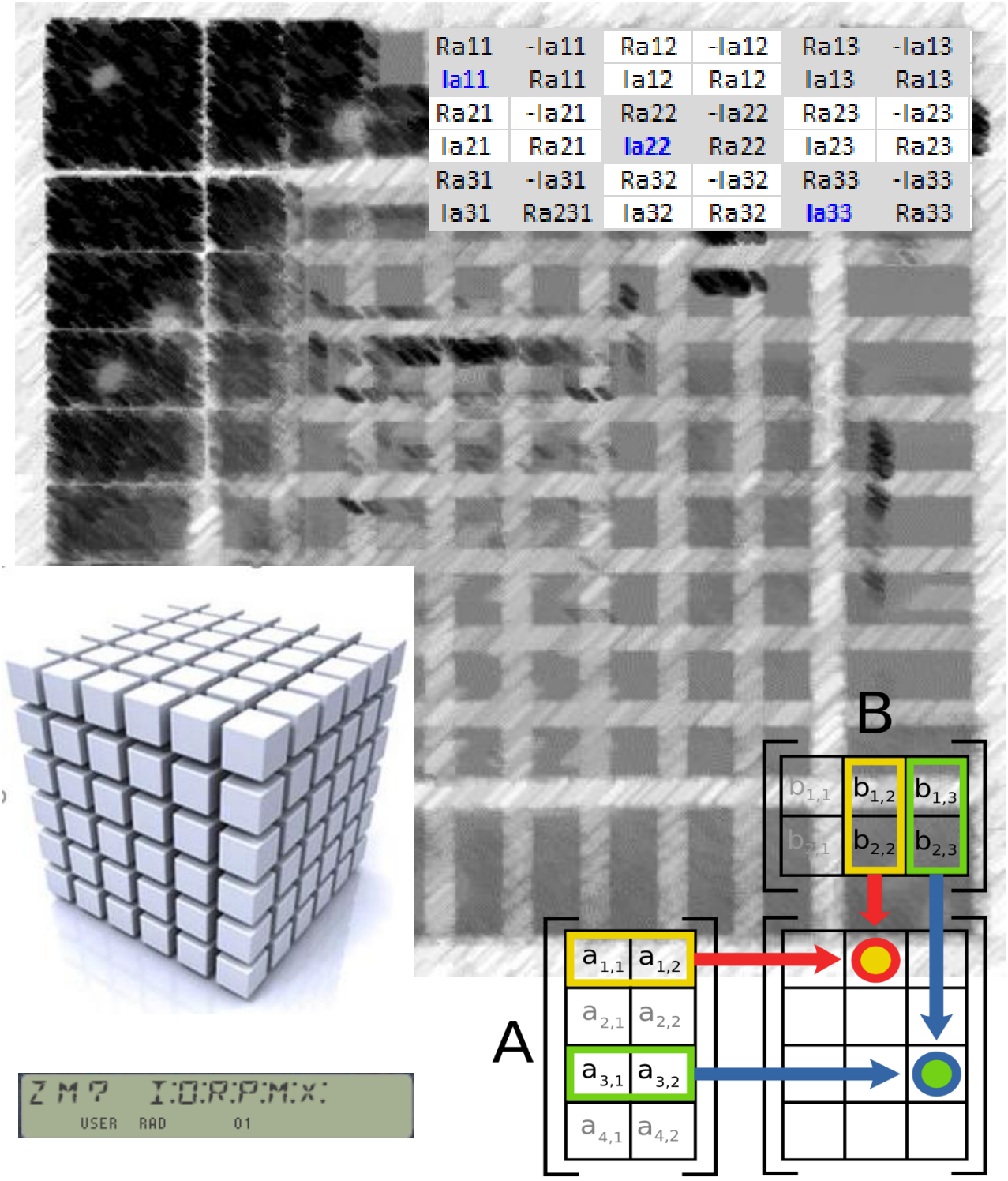
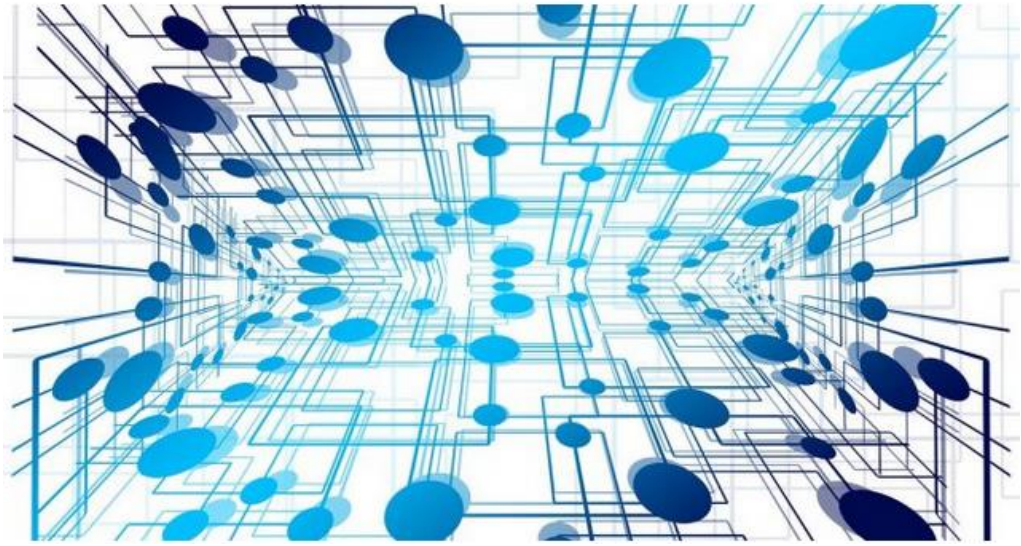
HP-41 Complex Matrix ROM

Extending the HP-41 SandMatrix – part II



Ángel M. Martin Cañas.

This compilation revision 2.2.7



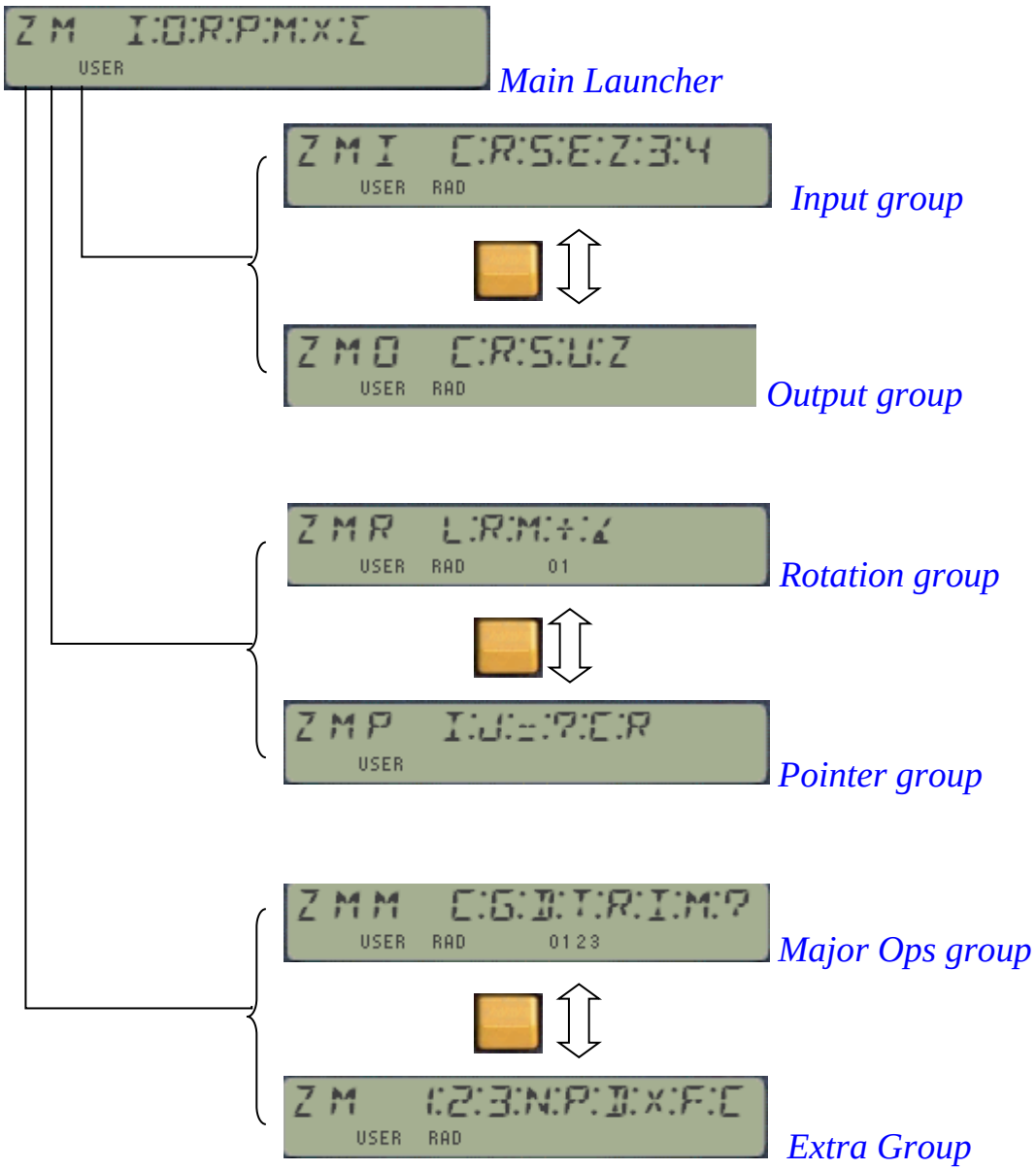
Front cover image taken from: *https://www.dreamstime.com/royalty-free-stock-photography-mathematics-background-image20849947*

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

COMPLEX_MATRIX 2C
HP-41 Module

Table of Contents

Function Launcher interdependencies.



Introduction: A Complex Chapter for the SandMatrix.

The SandMatrix module first and later the Advantage Math ROM have brought the state of the art to a further place, and now this module picks up where they left off extending the functionality to the Complex Matrices case. It is therefore another complementary extension to the SandMatrix, independent from the Advantage Math but that can be used in conjunction with it if so desired. As it can be expected it requires the 41Z to be plugged in the calculator to deal with the complex number math involved in many of the program included in the module – not only for the lower-level complex arithmetic but also using general-purpose utilities like ZSOLVE and others.

In this module you'll find:

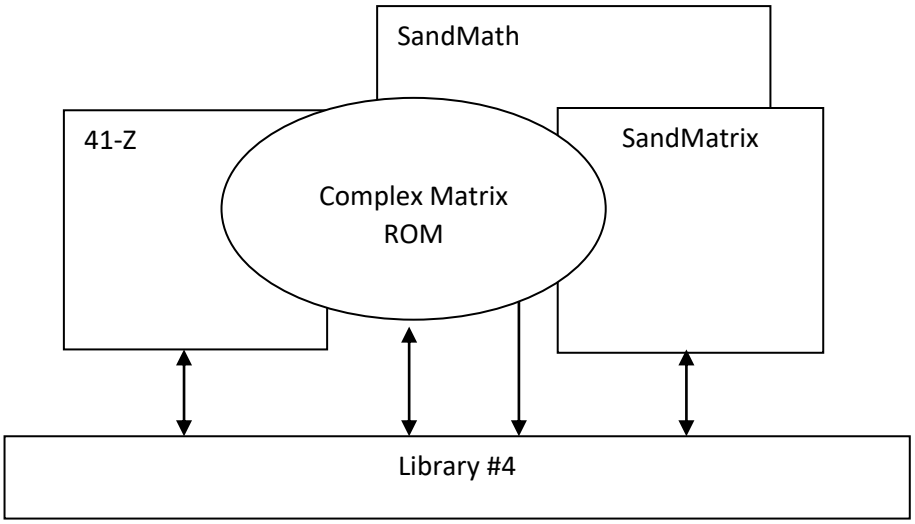
- More flexible Complex Matrix Editors that replace HP's **CMEDIT**, new pointer positioning and complex element storage & recall routines, as well as row/column exchange – all analogous to the real matrix case, but that will facilitate the writing of more complex programs.
- Missing functionality like Determinants and Matrix Transpose is also addressed in this ROM, with a couple of approaches to choose from depending on your preferences. Matrix Minors, Cofactors and Sub-matrices round off this part.
- A short update on Eigenvalues plus new sections on Matrix Element Rotation and other mappings.
- A **Function Launcher** grouping functions in logical groups

Module Dependencies.

As mentioned, you should have the 41Z and the SandMatrix plugged in the calculator. It is also recommended to have the SandMath plugged in as well, as several programs within the SandMatrix make use of its functions. If you plug those three, they will take 6 pages in the I/O bus, to add to the single page required by this ROM. This means that only one page will remain available in the external ports of the calculator, so use it judiciously – and what a better choice than the 'Advantage Math" ROM for a powerhouse Math configuration.

I strongly suggest the OS/X Module to be plugged as well, and if possible (i.e. without a printer) in page #6. The WARP_Core Module in page#7 will round up the perfect set.

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



ROM Function Tables.

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

XROM	Function	Description	Input	Author
13,00	**-CPX MTRX**	*Section Header*	*n/a*	*n/a*
13,01	**"CGTRP**	Matrix Conjugate Transpose	Name in ALPHA	*Ángel Martin*
13,02	**"CMINOR**	Complex Matrix Minor	iii,jii in X , Name in ALPHA	*Ángel Martin*
13,03	**CMDTM**	Complex DET w/ Minors	Name in ALPHA	*Ángel Martin*
13,04	**"CMTRP**	Complex Matrix Transpose	Name in ALPHA	*Ángel Martin*
13,05	**"SUBZM**	Reduces Matrix by index	iii,jjj in X, Name in ALPHA	*Ángel Martin*
13,06	**"^C33**	Creates 3x3 Cpx. Mat. Example	n/a	*Ángel Martin*
13,07	**"^C44**	Creates 4x4 Cpx. Mat. Example	n/a	*Ángel Martin*
13,08	**ΣZM _**	Function Launcher	"I:O:R:P:M;X"	*Ángel Martin*
13,09	**ZIJ?**	Recalls Complex Pointer	Name in ALPHA	*Ángel Martin*
13,10	**ZIJ=**	Sets Complex Pointer	iii,jii in X , Name in ALPHA	*Ángel Martin*
13,11	**ZI+**	Advances pointer one row	Name in ALPHA	*Ángel Martin*
13,12	**ZJ+**	Advances pointer one column	Name in ALPHA	*Ángel Martin*
13,13	**ZIMC**	Input Complex MAT by Cols	Name in ALPHA	*Ángel Martin*
13,14	**ZIMR**	Input Complex MAT by Rows	Name in ALPHA	*Ángel Martin*
13,15	**ZOMC**	Output Complex MAT by Cols	Name in ALPHA	*Ángel Martin*
136,16	**ZOMR**	Output Complex MAT by Rows	Name in ALPHA	*Ángel Martin*
13,17	**ZOUT**	Output Complex	Z in Y,X,	*Ángel Martin*
13,18	**>ZIJ**	Append Cpx. Element	iii,jjj in X	*Ángel Martin*
13,19	**"?CM**	Complex Matrix Test	Name in ALPHA	*Ángel Martin*
13,20	**-Z MATRX II**	*Section Header*	*n/a*	*n/a*
13,21	**ASWAP**	ALPHA Swap around comma	A,B in ALPHA	*Ángel Martin*
13,22	**E3/E+**	Builds pointer	1.00x	*Ángel Martin*
13,23	**"M=M?**	Equal Matrices Test	"M1,M2" in ALPHA	*Ángel Martin*
13,24	**"XZ>ZM**	Merges Matrix Components	"RE,IM,ZM" in ALPHA	*Ángel Martin*
13,25	**"Z+M**	Adds complex to Matrix	Z in {X,Y}, Name in ALPHA	*Ángel Martin*
13,26	**"Z*M**	Multiplies Matrix by Complex	Z in {X,Y}, Name in ALPHA	*Ángel Martin*
13,27	**"ZC<>C**	Swaps Complex Columns	iii,jjj in X, Name in ALPHA	*Ángel Martin*
13,28	**"ZEV22**	Eigenvalues of 2x2 Matrix	Name in ALPHA	*Ángel Martin*
13,29	**"ZEV33**	Eigenvalues of 3x3 Matrix	Name in ALPHA	*Ángel Martin*
13,30	**"ZEIGEN**	Eigenvalues using ZSOLVE	Name in ALPHA, zo in Z	*Ángel Martin*
13,31	**"ΣZIJJI**	Crossed-Elements products	Name in ALPHA	*Ángel Martin*
13,32	**"ZMDPS**	Diagonal Products Sum	Name in Alpha	*Ángel Martin*
13,33	**"ZM>XY**	Separates Matrix Components	"ZM,RE,IM" in ALPHA	*Ángel Martin*
13,34	**"ZMAT***	Element Products	"M1,M2,M3" in ALPHA	*Ángel Martin*
13,35	**"ZFNRM**	Frobenius Norm	Name in ALPHA	*Ángel Martin*
13,36	**"ZR<>R**	Swaps Complex Rows	iii,jjj in X, Name in ALPHA	*Ángel Martin*

13,37	**"ZCFTS**	Generates Cofactors Matrix	Name in ALPHA	*Ángel Martin*
13,38	**"ZMNRS**	Generates Minors Matrix	Name in ALPHA	*Ángel Martin*
13,39	**"ZMINV**	Inverse Matrix V2	Name in ALPHA	*Ángel Martin*
13,40	**"ZPCHR**	Characteristic Polynomial	Name in ALPHA	*Ángel Martin*
13,41	**-M-ROTATE**	*Section header*	*Checks if X is Odd*	*n/a*
13,42	**"?1X1**	Matrix Dimension Test	1,2 in X, Name in ALPHA	*Ángel Martin*
13,43	**"a<>b**	Swaps Real Elements	iii,jjj in X, "M1,M2" ALPHA	*Ángel Martin*
13,44	**"a>b**	Real Element Copy	iii,jjj in X, "M1,M2" ALPHA	*Ángel Martin*
13,45	**"Ca>b**	Complex Element Copy	iii,jjj in X, "M1,M2" ALPHA	*Ángel Martin*
13,46	**"M<)R**	Matrix Element Right Rotation	Name in ALPHA	*Ángel Martin*
13,47	**"MROT+**	Driver for "MROT	Prompts for data	*Ángel Martin*
13,48	**"MROT**	Matrix Elements Rotation	Data in XYZ, Name ALPHA	*JM Baillard*
13,49	**"ZM<)R**	Cpx. Mat. Element Right Rot	Name in ALPHA	*Ángel Martin*
13,50	**"ZMIRR**	Complex Matrix Mirror Image	Type in X, Name in ALPHA	*Ángel Martin*
13,51	**"ZMR90R**	Cpx. Mat. 90 deg Right Rot.	Name in ALPHA	*Ángel Martin*
13,52	**"ZMR90L**	Cpx. Mat. 90 deg Left Rot	Name in ALPHA	*Ángel Martin*
13,53	**"ZMROT+**	Driver for ZMROT	Prompts for Data	*Ángel Martin*
13,54	**"MSCRL$**	Matrix Scrolling	"MNAME,$" in ALPHA	*Ángel Martin*
13,55	**"SPLICE**	Data Registers Splicing	Register Range in X	*Ángel Martin*
13,56	**"SPLIT**	Data Registers Splitting	Register range in X	*Ángel Martin*
13,57	**"IMS"**	Input Matrix in Spiral	Name in ALPHA	*Ángel Martin*
13,58	**"OMS"**	Output Matrix in Spiral	Name in ALPHA	*Ángel Martin*
13,59	**"ZIMS"**	Input Cpx. Mtrx.inSpiral	Name in ALPHA	*Ángel Martin*
13,60	**"ZOMS"**	Output CCpx. Mtrx. In Spiral	Name in ALPHA	*Ángel Martin*
13,61	**"*ZEV**	Auxiliary for ZEIGEN	n/a	*Ángel Martin*
13,62	**"ZTRCE**	Alternative CMTRCE		*Ángel Martin*
13,63	**"ZTRNP**	Alternative CMTRP		*Ángel Martin*

1. SandMatrix Extensions for Complex Matrices

Although some of the SandMatrix functions support Complex Matrices, there's not a thorough coverage for these due to a justifiable lack of functionality. This module aims to closing that gap by providing a set of routines that can also be used in bigger programs to provide the missing features.

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. In the SandMatrix 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 $\begin{bmatrix} 1 & -2 \\ 2 & 1 \\ 3 & 4 \\ -4 & 3 \end{bmatrix}$

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

Besides these there are a few other SandMatrix functions that work equally right on either type of matrices, real and complex, provided that you follow the "double up" rule for the matrix dimension -

These functions are:

- **MIDN** Builds an Identity Matrix (Dimension 2m x 2n for complex)
- **MZERO** Clears all elements of a matrix
- **MZDG** Matrix Zero Diagonal
- **AIM** Anti-Identity Matrix
- **M^2** Multiplies a matrix by itself
- **MPWR** Matrix integer power

Notably the last two are a direct benefit for the complex matrix support included in **M*M,** and will play an important role in the complex matrix determinants programs described later on.

New Functions added for the Complex case

The table below shows most of the new functions added in this ROM, comparing the to the real-matrix analogous ones. You'll see some of them deal with the complex-pointer management, while others work as complete utilities for complex matrices.

Complex Matrix ROM	SandMatrixModule	Description
ZI+, ZJ+	**I+, J+**	Pointer advance
ZIJ?, ZIJ=	**MRIJA, MSIJA**	Pointer Store / Recall
CMR, CMS	**MR, MS**	Element Store / Recall
ZC<>C, ZR<>R	**C<>C, R<>R**	Column/Row Exchange
ZMNRM	**FNRM**	Frobenius Norm
Z+M	**MAT+** (w/ "X,A,B" in ALPHA)	Addition of Complex
Z*M	**MAT*** (w/ "X,A,B" in ALPHA)	Product by Complex
ZMAT*	**MAT*** (w/ "A,B,C" in ALPHA)	Element Product
M=M?	n/a	Equal Matrices Test
ZIMC, ZIMR, ZOMC, ZOMR	**IMC,IMR,OMC,OMR**	Complex Matrix Editor(s)
ZTRCE	**CTRCE, MTRACE**	Matrix Trace
CMDET, CMDTM	**MDET**	Determinant
CMTRP, CGTRP	**TRNPS**	Matrix Transpose
ZEIGEN	**EIGEN**	Eigenvalues by (Z)SOLVE
ZEV2	**EV2x2**	Eigenvalues of 2x2 Matrix
ZEV33	**EV3X3**	Eigenvalues of 3x3 Matrix
ZMDPS	**MDPS**	Diagonal products sum
ΣZIJJI	**ΣIJJI**	Crossed-els. Prods. sum
ZPCHR	**PCHAR**	Characteristic Polynomial

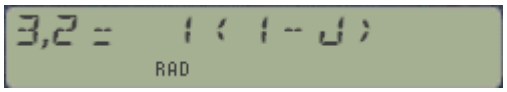
Complex Matrix ROM	Advantage Math ROM	Description
CMINOR, SUBZM	**MINOR, SUBM**	Minor and Submatrix
ZMROT, ZM<)R	**MROT, M<)R**	Matrix Element Rotation
ZM90R, ZM90L	**M90R, M90L**	Matrix 90 deg. Rotations
ZMIRR	**MMIRR**	Matrix Mirror Image
?CM	n/a	Matrix Complex Test
XY>ZM, ZM>XY	n/a	Complex Mat to Real, Imag

Note that some of the real matrix routines are not in the SandMatrix but in the "Advantage Math" ROM. This is the case for the element rotation and mirror image routines, as well as the minor/submatrix routines.

Note also that a couple of the new functions are included in the SandMatrix as well, either as main functions or as sub-functions – but are kept here as well to facilitate their usage and for consistency sake.

Complex Matrix Input / Output by Columns / Rows.

Analogous to the real counterparts **IMR**, **OMR**, **IMC**, **OMC**, you can use these routines to input and output the complex elements *of an existing matrix*, starting at the first element and sequentially either by rows or by columns. The display will show the **complex** index followed by the complex element value:

... 

, 

, etc...

With one exception, this set of routines replaces the **CMEDIT** program in the SandMatrix. The exception being cases where a complex column-matrix is used for linear systems with complex matrices; see the SandMatrix manual for details.

Routines for Element Storage and Recall.

These routines will use the complex pointer *iii,jjj* in register X to define the complex element position in the complex matrix. Note that the complex-pointer is always twice the real-pointer - which is the only one understood by the SandMatrix functions of course. The routines use auxiliary functions **ODD?** and **DIMERR** to halt the execution in case of not-even dimensions.

Program listing and Stack usage:

1	***LBL "CMR"**
2	ST+ X
3	**MSIJA**
4	RDN
5	**MRR-**
6	**MRR+**
7	X<>Y
8	RTN
9	***LBL "CMS"**
10	ST+ X
11	**MSIJA**
12	RDN
13	**MS**
14	**J-**
15	**I-**
16	**MSC+**
17	X<>Y
18	**MSR+**
19	CHS
20	**I-**
21	**MS**
22	CHS
23	X<>Y
24	RTN
25	***LBL "?CM"**
26	**DIM?**
27	**ODD?**
28	**DIMERR**
29	FRC
30	**I<>J**
31	**ODD?**
32	**DIMERR**
33	RDN
34	END

Routine / Stack	CMR		CMS	
	Input	Output	Input	Output
T:	T:	Z:	T:	scratch
Z:	Z:	Y:	Im(Zij)	T:
Y:	Y:	Im(Zij)	Re(Zij)	Im(Zij)
X:	Complex iii,jjj	Re(Zij)	Complex iii,jjj	Re(Zij)
L:	n/a	scratch	n/a	scratch
Matrix Pointer	n/a	iii,jjj	n/a	iii,jjj

Note that in the **CMR** case the complex pointer will be reset to the desired value, and that register T: will be lost – whereas for **CMS** the contents of stack register X: will be lost – so you'll need to use **ZIJ?** described later to recall the current pointer value.

Aided by the routines above the main Input/Output Programs are a simple affair managed by the flags 9 and 10 to control the out-of-bounds conditions.

As always, with flag 21 set the execution stops with every element display (function **ZOUT** takes care of that). This is a good resource to use for **ZOMR** and **ZOMC** with high TURBO settings on the 41CL.

For **ZIMR** and **ZIMC** the current element complex value will be shown in the prompt. Press R/S if you want to use it as is, or if you need to change it remember to enter the imaginary part first, then the real part separated by ENTER^ (same 41Z convention) – terminating the entry with R/S .

The program listing is shown below:

```
01  LBL "ZIMR"
02  E
03  GTO 03
04  LBL "ZIMC"
05  3
06  GTO 03
07  LBL "ZOMR"
08  0
09  GTO 03
10  LBL "ZOMC"
11  2
12  *LBL 03
13  X<>F
14  XROM "?CM"
15  E
16  E3/3+
17  *LBL 00
18  CMR
19  MRIJ
20  2
21  /
22  CLA
23  MPT
24  RDN
25  FS> 00
26  >"?"
27  ZOUT
28  MNAME?
29  FC? 00
30  GTO 01
31  STOP
32  MRIJ
33  2
34  /
35  CMS
36  I+
37  *LBL 01
38  FC? 01
39  J+
40  FS? 01
41  I+
42  FS? 10
43  GTO 01
44  FC? 09
45  GTO 02
46  FC? 01
47  I+
48  FS? 01
49  J+
50  *LBL 02
51  FC? 01
52  J+
53  FS? 01
54  I+
55  MRIJ
56  2
57  /
58  GTO 00
59  *LBL 01
60  CLD
61  END
```

Note that unlike the real pointer functions in the SandMatrix, the complex pointer positions for the first column and first row must be explicitly given, thus 1,001 is required in step 16 and cannot be replaced with "0". This holds true for any other combination that involves said first Column/row such as 1.003. 2.001, etc.

Lastly, the "**?CM**" routine is the *"is matrix Complex?"* check to discard non-complex matrices. You should be aware that said test is a *necessary but not sufficient* check, i.e. it checks that the matrix dimensions are even values, but it doesn't do any further testing re. the structure of the elements, such as ensuring that the real and imaginary parts of each complex element follow the proper rules. See below an example for such routine:

```
01 *LBL "?ZM"
02 XROM "?CM"
03 ,
04 MSIJA
05 *LBL 00
06 ZIJ?
07 VIEW X
08 ST+ X
09 MSIJ
10 MRR-
11 I-
12 MRC+
13 X#Y?
14 DIMERR
15 MRR+
16 I-
17 MRR+
18 CHS
19 X#Y?
20 DIMERR
21 FS? 09
22 I+
23 FC? 09
24 GTO 00
25 CLD
26 END
```

Routines for Complex Pointer positioning.

The routines below are the complex-case equivalent of functions **MRIJA**, **MSIJA**, **I+** and **J+** in the SandMatrix. Use them to retrieve the current complex-pointer, to advance it one position in either direction, or to reset it to a different value.

- The current complex-pointer retrieved by **ZIJ?** is always a valid pointer, regardless of the initial real-pointer of the matrix. This means that if needed **ZIJ?** will "nudge" the real pointer appropriately to always point at the proper complex counterpart.
- On the other hand, **ZIJ=** will check that the desired new complex pointer in X is within the bounds of the matrix, showing a DIM ERROR message if the input parameter is outside of it.
- Use **ZI+** and **ZJ+** to advance the complex pointer one position column-wise or row-wise respectively., i.e. **ZI+** increases the row value within the column and **ZJ+** increases the column value within the row. Note that like it's the case for the real-pointer functions **I+** and **J+**, these functions *don't return the current pointer to X*; you'll need to use **ZIJ?** for that.
- Advancing the complex-row or the complex-column really means moving two rows or two columns in the real matrix, and this is also done within the confines of the matrix dimensions. For example **ZJ+** will move the pointer to the *first element in the following row* if the last column had been reached - and conversely, **ZI+** will move the pointer to the *first element in the following column* if the last row had been exceeded, setting user flags 09 and 10 according to the same rules used for real matrices.
- To avoid straddled pointer settings (i.e. not-even real pointers), the first thing done by **ZI+** and **ZJ+** is to ensure that the initial complex pointer has a valid position using **ZIJ?**

Program listing:

```
01  LBL"ZJ+"
02  XROM "ZIJ?"
03  J+
04  FC? 09      ; within?
05  J+          ; yes, next
06  GTO 00      ; merge
07  LBL "ZI+"
08  XROM "ZIJ?"
09  I+
10  FC? 09      ; within?
11  I+          ; yes, next
12  *LBL 00
13  FC? 09      ; within?
14  RTN         ; yes, done.
15  FS? 10      ; outside?
16  RTN         ; yes
17  I+          ; no, nexti
18  J+          ; and next j
19  SF 09       ; set it up!
20  RTN         ; done.
21  LBL "ZIJ?
22  MRIJA
23  ODD?
24  I+
25  FRC
26  I<>J
27  ODD?
28  J+
29  RDN
30  MRIJ
31  2
32  /
33  RTN
34  LBL "ZIJ="
35  ST+ X
36  MSIJA
37  2
38  /
39  END
```

The table below shows the stack usage for the pointer routines:

Routine / Stack	ZI+, ZJ+, ZIJ?		ZIJ=	
	Input	Output	Input	Output
T:	T:	Lost	T:	Lost
Z:	Z:	Z:	Z:	Z:
Y:	Y:	Y:	Y:	Y:
X:	X:	X:	Complex iii,jjj	Complex iii,jjj
L:	n/a	scratch	n/a	scratch
Matrix Pointer	n/a	(iii+1),(jjj+1)	n/a	iii,jjj

Example Matrices C3X3 and C4X4

These routines simply load two example complex matrices in X-Memory – just a convenient and easy way to get you started with the examples included in the following sections of the manual. All you need to do is type the routine name and the corresponding matrix will be created.

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

and:

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

Note how the complex pointer is managed by the dedicated functions described before – no need to worry about the real and imaginary parts separately.

Program listing:

```
01 LBL "^C33"
02 "C3X3"
03 6,006
04 MATDIM
05 3
06 ENTER^
07 E
08 ENTER^
09 0
10 CMS
11 -5
12 ENTER^
13 2
14 XEQ 00
15 -1
16 ENTER^
17 7
18 XEQ 00
19 -2
20 ENTER^
21 4
22 XEQ 00
23 9
24 ENTER^
25 6
26 XEQ 00
27 4
28 ENTER^
29 -8
30 XEQ 00
31 -7
32 ENTER^
33 -3
34 XEQ 00
35 2
36 ENTER^
37 3
38 XEQ 00
39 6
40 ENTER^
41 -1
42 *LBL 00
43 XROM "ZJ+"
44 XROM "ZIJ?"
45 VIEW X
46 CMS
47 END
```

Totally Unnecessary Examples. ;-)

The routines below are an alternative implementation of the MCODE function **M*M** and the focal routine **M^2** included in the SandMatrix. They don't really improve on the original ones (and certainly are much slower!) but have been included as academic examples of utilization for the pointer and element storage & recall functions.

The matrix multiplication definition is well known, and can be summarized by the expression below: If **A** is an $m \times n$ matrix and **B** is an $n \times p$ matrix, the *matrix product* **C** = **AB** (denoted without multiplication signs or dots) is defined to be the $m \times p$ matrix such that

$$c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + \cdots + a_{in}b_{nj} = \sum_{k=1}^{n} a_{ik}b_{kj},$$

Program Listing:

1	LBL "ZM*M"	A,B,C	38	ASWAP	A,B,C	
2	DIM?	2m,00(2n)	39	RCL 03	i,00m	
3	STO 00	2m,00(2n)	40	INT	i	
4	2		41	RCL 01	k,00n	
5	/	m,00n	42	INT	k	
6	E3/E+	1,00m00n	43	STO 00		
7	STO 03	1,00m	44	I<>J	0,00k	
8	ASWAP	B,C,A	45	+	i,00k	
9	DIM?	2n,00(2r)	46	XROM "CMR"		
10	2		47	ZENTER^	pust it	
11	/	n,00r	48	ASWAP	B,C,A	
12	FRC		49	RCL 02	j,00r	
13	STO 02	0,00r	50	INT		
14	LASTX	n,00r	51	I<>J	0,00j	
15	INT		52	RCL 00	k	
16	i<>J		53	+	k,00j	
17	STO 01	0,00n	54	XROM "CMR"		
18	ASWAP	C,A,B	55	Z*	aik.bkj	
19	ST+ X	0,00(2r)	56	Z+	partial result	
20	RCL 00	2m,002r	57	ZENTER^	push it	
21	INT	2m	58	ASWAP	C,A,B	
22	+	2m,00(2r)	59	ISG 01	NEXT k	
23	MATDIM	*create resut matri*	60	GTO 01		
24	LBL 03	M1 Rows	61	ZRDN	pop result	
25	RCL 02	x,00r	62	RCL 03	i,00m	
26	FRC		63	INT	i	
27	E		64	RCL 02	j,00r	
28	+	1,00r	65	INT	j	
29	STO 02	1,00r	66	I<>J	0,00c	
30	LBL 02	M2 rows	67	+	i,00j	
31	RCL 01	x,00n	68	XROM "CMS"	save C element	
32	FRC		69	ISG 02	NEXT J	
33	E		70	GTO 02		
34	+		71	ISG 03	NEXT I	
35	STO 01	1,00n	72	GTO 03		
36	CLZST	reset Z-stack	73	ASWAP	A,B,C	
37	LBL 01	M1 cols = M2 rov	74	END	done.	

Program remarks:

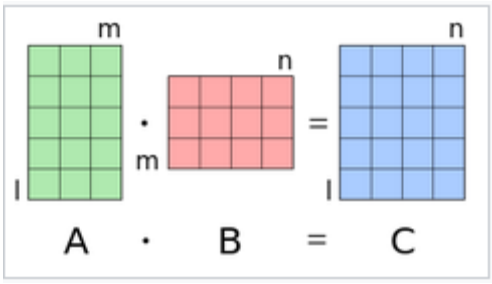
The initial part of the program creates the result matrix and resets all its elements to zero. It then prepares the ranges for each of the three counters used for index and repeats the three loops accordingly. LBL 01 is the lowest-level loop, used to compute each element of the result matrix. Nothing sophisticated here; we use **CMR** twice to recall the appropriate factors from the input matrix and **CMS** to store the resulting element in the result matrix. Note the use of the complex stack to hold the partial results of the intermediate multiplications and sums, very convenient as it doesn't interfere with the data registers in memory or the real stack.

The program uses data registers R00 – R03, and expects the matrix names in ALPHA separated by commas: "M1, M2, M3". Note that M3 cannot be any of the operator matrices (so it's not an in-place implementation). R01 holds the value of index k, R02 the value of index j, and R3 the value of index i. The routine needs to prepare their ranges based on the matrix dimensions, and then proceeds with the algorithm as defined above.

Corollary: Matrix Square product.

The routine below is an almost-trivial driver for **ZM*M** that computes the square power of a given squared matrix. Note the use of scratch matrix "#" to temporarily hold the result matrix, before it is copied back to the source matrix – and eventually purged from memory.

1	LBL "ZM^2"	*MNAME*
2	XROM "?CM"	
3	ASTO 04	
4	>","	*MNAME,*
5	ASTO X	
6	DIM?	
7	"!="	
8	MATDIM	
9	CLA	
10	ARCL Y	*MNAME,*
11	ARCL Y	*MNAME,MNAME,*
12	>"!="	*MNAME,MNAME,#*
13	XROM "ZM*M"	
14	"!=,"	
15	ARCL 04	*#,MNAME*
16	CLST	
17	MMOVE	
18	PURFL	
19	MNAME?	*MNAME*
20	END	



Example: calculate the square of matrix [C3X3]. The result is given below:

1,1=-38-J64	2,1= 104+J78	3,1= 79-J25
1,2= 97	2,2=-79+J80	3,2=-56+J56
1,3= 13+J111	2,3=-74-J118	3,3=-95-J62

If nothing else these examples should have increased your appreciation for the MCODE function **M*M**, whose only caveat is not supporting in-place operation but certainly works much faster and without any data registers requirements – and not needing the 41Z either for the complex arithmetic.

Complex Transposed Matrix. {CMTRP , ZTRNP }

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 program steps / byte count.

For a complex element pointer (i,j) the relationship with the individual matrix 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 the 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 perform 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 12 to negate the imaginary parts – with no impact to the **CMR** and **CMS** subroutines.

Example.- Transpose the 3x3 complex matrix shown below (you can use '^**C33**" to create it), and use **ZOMR** to review the result.

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

Corollary: Conjugate-Transposed Matrix. { CGTRP , CMS , CMR }

In mathematics, the conjugate transpose or **Hermitian** transpose of an m-by-n matrix [A] with complex entries is the n-by-m matrix $[A]^H$ obtained from [A] by taking the transpose and then taking the complex conjugate of each entry. Thus, the conjugate transpose of an m × n matrix [A] is formally defined by:

$$\boldsymbol{A}^{\mathrm{H}} = \left(\overline{\boldsymbol{A}}\right)^{\mathrm{T}} = \overline{\boldsymbol{A}^{\mathrm{T}}}$$

The routine does a direct utilization of the element-based method shown before; so there's not earth-shattering math involved in it. It does, however, provided the basis for a pair of sub-routines to save and recall a complex element in the matrix using the complex index of the element. These subroutines will be used by other, more complex programs described later on.

```
01  *LBL "CGTRP"
02  "/-,#"
03  DIM?
04  I<>J
05  ASWAP
06  MATDIM
07  ASWAP
08  E
09  E3/E+
10  CF 00
11  *LBL 01
12  CF 01
13  ENTER^
14  CMR
15  X<>Y
16  CHS
17  X<>Y
18  J+
19  FS? 09
20  SF 01
21  FS? 10
22  SF 00
23  ASWAP
24  RCL Z
25  I<>J
26  CMS
27  ASWAP
28  FS? 00
29  GTO 02
30  RCL Z
31  FS? 01
32  INT
33  E
34  E3/E+
35  FC? 01
36  FRC
37  +
38  GTO 01
39  *LBL 02
40  ASWAP
41  CLST
42  MMOVE
43  PURFL
44  MNAME?
45  RTN
46  *LBL "CMR"
47  ST+ X
48  MSIJA
49  RDN
50  MRR-
51  MRR+
52  X<>Y
53  RTN
54  *LBL "CMS"
55  ST+ X
56  MSIJA
57  RDN
58  MS
59  J-
60  I-
61  MSC+
62  X<>Y
63  MSR+
64  CHS
65  I-
66  MS
67  CHS
68  X<>Y
69  END
```

An alternative version using the pointer routines is also listed below. It's simpler but slower (two sweeps are needed), it requires data registers R00 & R01, plus the 41Z module as well:

```
1  LBL "ZHRMT"
2  XROM "CTRNP"
3  E
4  E3/E+
5  ZIJ=
6  *LBL 00
7  ZIJ?
8  STO 00
9  CMR
10 ZCONJ
11 RCL 00
12 CMS
13 ZJ+
14 FC? 10
15 GTO 00
16 END
```

Complex Matrix Determinants. { CMTRC , CMDET }

Note: Superseded by the ZPCHR method described in section 3.e

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 for orders 2, 3, and 4 - 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 **ZIMR** routine - which expects the matrix name in ALPHA. It also expects the matrix already created, using 2n x 2n as dimension - with "n" being the complex 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 such can't exceed 6 characters).

As you can see there are numerous 41Z functions –needed 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 (you can use "C4X4" to create it):

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

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 very 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.

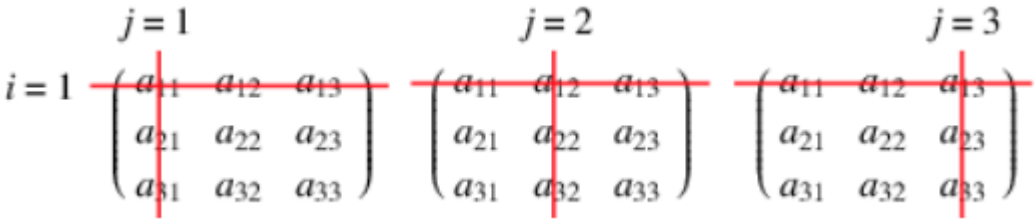
Program Listing.

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	

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 sub-matrix formed by deleting the i-th row and j-th column. This number is often denoted Mi,j. The (i,j) cofactor is obtained by multiplying the minor by: (-1)^{i+j}.



Two programs are included, one for Real matrices (not limited in order, courtesy of **MDET**) and another for Complex Matrices – only up to degree 5, due to the restriction imposed by **CMDET**. The programs are a good example of use of the matrix 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 example:

```
    [ 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. Remember also that the 41Z Module needs to be plugged for the complex determinant calculation.

The solutions are:

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

Corollary: Complex Determinant of order Five. {CMDTM}

An extension for 5x5 complex matrix determinants is included in the Complex Matrix ROM. It uses the complex minors (available up to order 4) to obtain the result. Note that since **CMINOR** uses **CMDET** in turn, therefore is not practical at all for orders 4 and below – which they'd better use a direct call to **CMDET**. You *can* do it of course, but it'll definitely be much slower.

```
01  LBL "CMDTM"
02  DIM?
03  2
04  /
05  FRC
06  E
07  +
08  STO 03
09  CLST
10  *LBL 00
11  ZSTO
12  2
13  RCL 03
14  XROM "CMINOR"
15  ZENTER^
16  RCL 03
17  CMR
18  RCL 03
19  INT
20  LASTX
21  I<>J
22  +
23  ZCHSX
24  Z*
25  ZRC+
26  2
27  ISG 03
28  GTO 00
29  ZAVIEW
30  END
```

The determinant is obtained as the sum of the elements multiplied by the corresponding co-factors (a sign-modified version of each minor). For the example matrix shown below (C3X3) these are as follows:

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

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

Determinant: (1+3i).(-28+31i) + (2-5i).(44+18i) + (7-i).(-29+71i)= -75 + 289i

Example of a Complex 5x5 Matrix in all its redundant glory:

Ra11	-Ia11	Ra12	-Ia12	Ra13	-Ia13	Ra14	-Ia14	Ra15	-Ia51
Ia11	Ra11	Ia12	Ra12	Ia13	Ra13	Ia14	Ra14	Ia15	Ra51
Ra21	-Ia21	Ra22	-Ia22	Ra23	-Ia23	Ra24	-Ia24	Ra52	-Ia52
Ia21	Ra21	Ia22	Ra22	Ia23	Ra23	Ia24	Ra24	Ia52	Ra52
Ra31	-Ia31	Ra32	-Ia32	Ra33	-Ia33	Ra34	-Ia34	Ra53	-Ia53
Ia31	Ra31	Ia32	Ra32	Ia33	Ra33	Ia34	Ra34	Ia53	Ra53
Ra41	-Ia41	Ra42	-Ia42	Ra43	-Ia43	Ra44	-Ia44	Ra54	-Ia54
Ia41	Ra41	Ia42	Ra42	Ia43	Ra43	Ia44	Ra44	Ia54	Ra54
Ra51	-Ia51	Ra52	-Ia52	Ra53	-Ia53	Ra54	-Ia54	Ra55	-Ia55
Ia51	Ra51	Ia52	Ra52	Ia53	Ra53	Ia54	Ra54	Ia55	Ra55

Sub-Matrices, Minors & Cofactor Matrices. {SUBZM , ZMNRS , ZCFTS }

Three other routines are also included to reduces Sub-matrix from a given one – replacing the original - and to build the minors and cofactors matrices of a given matrix - creating a new matrix of the same dimension where all elements will be the corresponding minor or cofactor for its position, like it was shown in the example below.

- Sub-matrices **will replace the original matrix**. Make sure you make a copy of it prior to the execution of **SUBZM** if you need the original matrix to be available for other purposes
- The minors and cofactors matrices are named "ZMNRS" to differentiate it from the real-case equivalent function in the Advantage Math ROM, which is named "MINORS"
- They are not to be confused with the routines available in the Advantage Math ROM, which are intended for the real-matrix case. Refer to the table below for details:

Complex Case (CPX MTRX ROM)	Function	Matrix Name
Element Minor (iii,jjj)	CMINOR	Result in {X,Y}
Sub-Matrix (iii,jjj)	SUBZM	Replaces original matrix
Minors Matrix	ZMNRS	ZMNRS
Cofactors Matrix	ZCFTS	ZMNRS

Real Case (Advtg Math ROM)	Function	Matrix Name
Element Minor (iii,jjj)	MINOR	Result in X
Sub-Matrix (iii,jjj)	SUBM	Replaces original Matrix
Minors Matrix	MINORS	MINORS
Cofactors Matrix	CFACTS	MINORS

Program Listing and data requirements:

Uses User flags 00-02 and data Registers {R00 – R04}. Matrix name is stored in R03.

```
01  LBL "ZMNRS"
02  CF 02 ; flags case
03  GTO 02
04  LBL "ZCFTS"
05  SF 02      ; flags case
06  *LBL 02
07  XROM "?CM"
08  ASTO 03
09  DIM?
10  "ZMNRS"
11  MATDIM
12  CLA
13  ARCL 03
14  E
15  E3/E+
16  ZIJ=
17  *LBL 04
18  ZIJ?
19  STO 04
20  XROM "CMINOR"
21  FC? 02
22  GTO 04
23  RCL 04
24  INT
25  LASTX
26  FRC
27  I<>J
28  +
29  ZCHSX
30  *LBL 04
31  "ZMNRS"
32  RCL 04
33  CMS
34  CLA
35  ARCL 03
36  ZJ+
37  FC? 10
38  GTO 04
39  END
```

Note the usage of **ZJ+** to sweep all elements in the original matrix (whose name is saved in R03), and the successive calculation of the minor or cofactor depending on the status of user flag 00.

The program below shows the current version for **CMINOR** and **SUBZM** – slightly tweaked revision of the original program listed before thanks to the utilization of some of the new routines.

```
01 LBL "SUBZM"
02 SF 01     ; flags case
03 GTO 01
04 LBL "CMINOR"
05 CF 01     ; flags case
06 *LBL 01
07 STO 00  ; iii,jjj
08 ASTO 03
09 ">",#"     ; aux matrix
10 MAT=
11 DIM?
12 2,002
13 -
14 "="       ; minor mat
15 MATDIM
16 MZERO
17 "#"       ; minor mat
18 RCL 00  ; iii,jjj
19 I<>J      ; jjj,iii
20 SF 00     ; do columns
21 XEQ 02
22 RCL 00  ; iii,jjj
23 CF 00     ; do rows
24 XEQ O2
25 CLST      ; prepare pts
26 3,003     ; real anchor
27 "#,="     ;from, to
28 MMOVE
29 PURFL     ; kills "#"
30 FS? 01    ; sub-mat?
31 GTO 01    ; yes, skip
32 MNAME?
33 XROM "CMDET"
34 PURFL     ; purge
35 "#"
36 CLA
37 ARCL 03
38 RTN
39 *LBL 01 ; submatrix
40 "=,"
41 ARCL 03 ; "=,A"
42 MAT=    ; override it
43 PURFL   ; kills "="
44 MNAME?
45 RTN
46 *LBL 02 ; subroutine
47 ST+ X    ; complex pt
48 STO 02  ; save in R02
49 XEQ 03  ; swap one
50 RCL 02  ; pointer
51 E
52 E3/E+   ; 1,001
53 -       ; previous
54 *LBL 03 ; swap two
55 RCL 00  ;iii,jjj
56 FS? 00  ;cols?
57 I<>J      ; yes
58 INT
59 DSE X   ; one less
60 X=0?    ; first one?
61 RTN     ; yes, done.
62 X<>Y    ; no, go on
63 INT
64 ENTER^
65 ENTER^
66 2
67 -
68 I<>J
69 +
70 *LBL 00
71 FS? 00  ; cols?
72 C<>C    ; bubble up
73 FC? 00  ; rows?
74 R<>R    ; bubble left
75 2,002   ; next pair
76 -
77 DSE Y   ; next?
78 GTO 00 ; yes, repeat
79 END     ; all done
```

Program remarks:

- It uses user flags 00 and 01, and data registers R00 – R03 for the matrix name
- The program uses two scratch matrices named "#"and "=". The first one is a direct copy of the given matrix to perform the alterations on it without modifying the original. The second one is a reduced matrix (one order smaller) that will hold the minor to calculate the determinant.
- The main subroutine LBL 02 does a "bubble" swap of the corresponding columns and rows – as per the complex pointer iii,jjj input value. The complex matrix requires two real columns and rows to be swapped, moving them to the upper-left corner of the matrix. Once completed the arrangement is ready for a clean "extraction" starting at real pointer 3,003 and into the destination minor matrix "=" (MMOVE in line 28)
- Note that **SUBZM** copies the resulting sub-matrix into the original, therefore it is replaced by it.
- Both auxiliary matrices are purged by the program on completion, and the original matrix (or sub-matrix) is left as the selected one.

Complex Matrix Trace. {CMTRC , ZMTRC }

The Complex Matrix ROM includes two routines to calculate the matrix trace:

The **CMTRC** routine uses a *direct* approach, summing the imaginary parts of the elements in the diagonal and using **MTRACE** to obtain the double of the sum of the real parts – as consequence of the particular way the data elements are arranged.

The **ZMTRC** uses an *indirect* approach, significantly slower than the previous one but interesting from the academic point of view because it can also be used for other needs. The idea is to separate the real and imaginary parts into two independent matrices, [Re(Z)] and [Im(z)]so that it verifies:

$$[Z] = [Re(Z)] + j\,[Im(Z)]$$

Interestingly enough this alternative method is less demanding on data storage requirements (half the size of the "standard" method), as each of the auxiliary matrices are of dimension (m x n) for a total of (2.m.n) elements instead of (2m.2n) = (4.m.n)

Obviously using the separate approach, the **MTRACE** function can be used on both the real and imaginary component matrices, and the trace is given by:

$$\text{trace}([Z]) = \text{trace}([Re(Z)]) + j\,\text{trace}([Im(Z)])$$

The downside is of course that we need a routine to separate the matrix first, which is going to take time to execute and code space in the ROM. Nevertheless, it's well worth the cost, as we'll be using it for other purposes besides the trace, such up matrix transposition and element rotations.

Program Listings.-

```
01 LBL "CMTRC"
02 MTRACE       ; 2x Real part
03 2
04 /            ; real part
05 ,
06 MSIJA        ;first element
07 I+           ; next row
08 MRC+         ;get Im(z11)
09 *LBL 00
10 I+           ;move down
11 J+           ; move left
12 J+           ; and left again
13 MRC-         ; get Im(zij)
14 +            ; add to partial
15 FC? 09       ; end of matrix?
16 GTO 00       ; no, do next
17 RCL Z        ; yes, real part
18 ZAVIEW       ; show result
19 MNAME?       ; recall MNAME
20 RTN          ; done.
21 LBL " ZTRCE"
22 ASTO 04      ; saves MNAME
23 >",#,="      ; scratch matx
24 XROM "ZM>XY"
25 ASWAP        "#,MNAME,="
26 MTRACE
27 ASWAP        "=,#,"MNAME"
28 MTRACE
29 X<>Y
30 PURFL        ; kills "="
31 "#"
32 PURFL        ; kills "#"
33 ZAVIEW       ; show result
34 CLA
35 ARCL 04      ; recall MNAME
36 END          ; done.
```

It must be said that separating real and imaginary components doesn't save any room when the original matrix is left around – and for a while the three always exist at once of course.

Separation into Real and Imaginary Components. {ZM>XY , XY>ZM }

Here are the routines for the separation into the real & imaginary parts, and the reverse operation to re-assembly the complex matrix from its two components. The routine will create the separate components if they didn't already exist.

Input: Names of the three matrices in the ALPHA register, "ZM, REAL,IMAG" for the separation routine and "REAL,IMAG,ZM" for the reverse. There are several calls to **ASWAP** to re-arrange ALPHA as needed – let the swap dance begin!

```
 1  LBL "ZM>XY"; ZM,RE,IM
 2  DIM?          ; 2m x 2n
 3  2
 4  /             ; m x n
 5  STO 02
 6  ASWAP         ; RE,IM,ZM
 7  MATDIM
 8  ASWAP         ; IM,ZM,RE
 9  MATDIM
10  ASWAP         ; ZM,RE,IM
11  XROM "?CM"
12  RCL 02        ; m x n
13  E3/E+         ; 1,00m(00n)
14  STO 02        ; r,00m
15  LASTX         ; m x n
16  *LBL 10       ; do columns
17  FRC           ; 0,00n
18  E
19  +             ; 1,00n
20  STO 01
21  *LBL 00       ; do rows
22  RCL 02        ; r,00m
23  INT           ; r
24  RCL 01        ; c,00m
25  INT           ; c
26  I<>J          ; o,ooc
27  +             ; r,00c
28  STO 00        ; current pt
29  CMR           ; recalls Zcr
30  XEQ 04        ; do the trick
31  ISG 01        ; next col
32  GTO 00        ; do next
33  RCL 01        ; reset counter
34  ISG 02        ; next row
35  GTO 10        ; do next
36  RTN           ; done.
37  *LBL 04
38  ASWAP         ; RE,IM,ZM
39  RCL 00        ; r,00c
40  MSIJA
41  X<>Y          ; Re(z)
42  MS
43  X<>Y          ; r,00c
44  ASWAP         ; IM,ZM,RE
45  MSIJA
46  RCL Z         ; Im(z)
47  MS
48  ASWAP         ; ZM,RE,IM
49  END
```

```
 1  LBL "XY>ZM"  ; RE,IM,ZM
 2  DIM?         ; m x n
 3  ASWAP        ; IM,ZM,RE
 4  DIM?         ; m x n
 5  ASWAP        ; ZM,RE,IM
 6  ST+ X        ; 2m x 2n
 7  MATDIM
 8  ASWAP        ; RE,IM,AM
 9  ,
10  MSIJA
11  *LBL 00
12  MRIJA        ; get current pt
13  MR           ; Re(z) to X
14  ASWAP        ; IM,ZM,RE
15  X<>Y         ; i,00j
16  MSIJA
17  MR           ; Im(z)
18  X<> Z        ; Re(z) to X
19  ASWAP        ; ZM,RE,IM
20  RCL  Z       ; i,00j
21  CMS          ; save cpx. elmt.
22  ASWAP        ; RE,IM,ZM
23  MRIJA        ; re-focus matrix!
24  J+           ; next column
25  FC? 10       ; ouside?
26  GTO 00       ; no, do next
27  END          ; yes, done!
```

Program remarks:

Note how in the separation routine **ZM>XY** we haven't used the pointer routines (**ZIJ?**,**ZJ+**, etc.) to navigate the matrix, rather the implementation uses two counters to move along the columns (in R01) and the rows (in R02). This requires prepping the ISG values in advance but it's faster than the complex pointer-based alternative. The other advantage is that ALPHA is not touched up at all, which becomes critical in these cases holding three matrix names at once.

The reverse action in **XY>ZM** (merging real and imaginary parts into a complex matrix) is simpler because here the pointer functions work on the real matrices, and not on the complex one. Thus there's no need for row and column counters since we use user flag 10 as termination criteria. Note however the careful use of **MSIJA** and **MRIJA** (with the "A") every time we need to recall an element from the two source matrices. Finally, the complex pointer value for [ZM] equals the real one in [RE] and [IM], given the doubled-up dimensions of the complex matrix compared to the real ones.

Corollary: Matrix Transpose revisited.

The example below is an alternative to the **CMTRP** program that also uses the matrix separation routine to apply the standard **TRNPS** function to each of the components, rebuilding the complex matrix at the end. It's probably slower than the routines described already for the transposition, but it may be interesting for large sizes.

Program Listing:

```
01    *LBL "ZTRNP"      MNAME in ALPHA
02    ASTO 04           ;  saves MNAME
03    >",#,="           ;  scratch matrix
04    XROM "ZM>XY"
05    ASWAP             ; "#,=,MNAME,="
06    TRNPS
07    ASWAP`            ; "=,MNAME,#"
08    TRNPS
09    ASWAP             ; "MNAME,#,="
10    ASWAP
11    XROM "XY>ZM"      ; rebuild the matrix
12    PURFL             ; get rid of components
13    "#"
14    PURFL
15    CLA               ; recall name
16    ARCL 04
17    END
```

2. Further Operations with Complex Matrices

This section deepens on the applications included in the SandMatrix, properly extended to support complex matrices. It is heavily reliant on the 41Z so don't forget to have it plugged in as well.

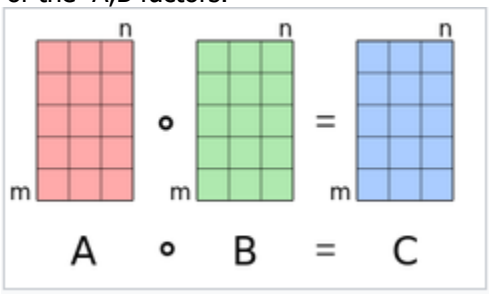
Addition and multiplication by complex values.

Completing the original matrix function set, you can use this pair of routines to:

- add one complex value to all elements of a complex matrix , and
- multiply all elements of a complex matrix by a complex value
- Compute the Hadamard product (element-wise) of two matrices

The first two **Z+M** and **Z*M** are particular examples of the real-matrix counterpart functions **MAT+** and **MAT*** when they're used with "X" in ALPHA to denote the X-register. The complex value is expected in {Y,X} and the matrix name should be in ALPHA.

The third one **ZMAT*** is the complex counterpart of the SandMatrix function **MAT***, which computes the element-wise multiplication of two matrices of equal dimensions. Like in the real function the three matrix names are expected in ALPHA, separated by commas. "A,B,C" – where C may also be one of the A,B factors.



Simple enough but having them available facilitates the writing of more complex routines, therefore their inclusion as separate entries in the FAT.

Example. Multiply the example matrix C3X3 by the imaginary unit j - i.e. do a 90 degree counter-clockwise rotation of the original complex elements.

Assuming C3X3 is already loaded in X-memory (if not just call "**^C33**") , we type:

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

1, ENTER^, 0, ALPHA, "C3X3", ALPHA, XEQ "Z*M"
Then SF 21, XEQ "ZOMR" to review the results, which should look as follows:

```
        [-3+j1  5+j2    1+j7]
i.[A] = [ 2+j4  -9+j6   -4-j8 ]
        [ 7-j3  -2+j3   -6-j1 ]
```

Registers, flags, etc.

These routines share common core block to sweep thru all matrix elements. The navigation does not use the new pointer functions, but a more direct, faster method instead. This requires using indexes in data registers R01 and R02, plus another register (R00) to hold the current pointer for the elements being processed. Data register R03 holds the subroutine index value, used by XEQ IND 03 in program step #57. Finally, user flag 01 is used for the **Z+M** and **Z*M** routines, but no scratch matrices are needed.

Program listing:

```
01 LBL "ZNRM"   ;MNAME in ALPHA
02 CLSTZ        ; clear Z=stack
03 5            ; subroutine index
04 XEQ 02       ; needs to return
05 ZRDN         ; drop result
06 SQRT         ; square root
07 MNAME?       ; select matrix
08 RTN          ; done.
09 LBL 05       ; does F-Norm
10 ZMOD         ; modulus
11 X^2
12 Z+           ; partial result
13 ZENTER^      ; push in Z-stack
14 RTN
15 LBL "ZMAT*"  "M1,M2,M3"
16 DIM?         ; M1 dimension
17 ASWAP        ; M2,M3,M1
18 DIM?         ; M2 dimension
19 X#Y?         ; different?
20 DIMERR       ; show error
21 ASWAP        ; M3,M1,M2
22 MATDIM       ; créate matrix
23 ASWAP        ; M1, M2,M3
24 3            ; subroutine index
25 GTO 02       ; merge with core
26 LBL "Z*M"
27 SF 01        ; flag case
28 GTO 01       ; merge case
29 LBL "Z+M"
30 CF 01        ; flag case
31 LBL 01       ; common
32 ZRPL^        ; fill complex stack
33 2            ; subroutine index
34 LBL 02       ; core block
35 STO 03       ; save index
36 DIM?         ; m,00n
37 XROM "?CM"   ; check if complex
38 2
39 /            ; complex dim
40 E3/E+
41 STO 02       ; 1,00m00n
42 LASTX        ; m,00n
43 LBL 10       ; row loop
44 FRC          ; 0,00n
45 E
46 +            ; 1,00n
47 STO 01
48 LBL 00       ; column loop
49 RCL 02       ; r,00m
50 INT          ; r
51 RCL 01       ; j,00n
52 INT          ; j
53 I<>J         ; 0,00j
54 +            ; r,00j
55 STO 00       ; complex pointer
56 CMR          ; recall element
57 XEQ IND 03   ; call subroutine
58 ISG 01       ; next column
59 GTO 00
60 RCL 01       ; recall pointer
61 ISG 02       ; next row
62 GTO 10
63 RTN          ; all done
64 LBL 02       ; does Z+M or Z*M
65 FC? 01       ; addition?
66 Z+           ; yes
67 FS? 01       ; product?
68 Z*           ; yes
69 RCL 00       ; recall pointer
70 CMS          ; save in result matrix
71 RTN          ; done.
72 LBL 03       ; does MAT*
73 ZENTER^      ; pushes M(k,j)
74 ASWAP
75 RCL 00       ; current pointer
76 CMR          ; recall element
77 Z*           ; complex multiply
78 ASWAP        ; rotate string
79 RCL 00       ; current pointer
80 CMS          ; store element
81 ASWAP
82 END
```

Corollary: Frobenius Norm. { **ZMNRM** }

Using the same definition as for real matrices the **ZNRM** routine is part of the same core routines, and therefore is included i this section.

$$\|A\|_{\mathrm{F}} = \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n}|a_{ij}|^2} = \sqrt{\operatorname{trace}(A^* A)}$$

where **A*** denotes the conjugate-transpose of A , and interestingly enough the expression above shows that *the trace of such matrix product must be a real number!*

Note that the Frobenius norm is obviously a real value also for complex matrices, by virtue of the modulus function.

Example: obtain the norms of our beloved example matrices C3X3 and C4X4:

||C3X3|| = 20.34698995
||C4X4|| = 11.00000000

Program remarks:

Note how the **ZMNRM** routine is also listed in the previous page, from program step #01 to #14 . It shares the same central core used by the Hadamard multiplication and element sum/product routines described just before, and therefore it is placed in this section even if strictly speaking it's not related to the same functionality.

The routine uses the complex arithmetic function **Z+** in step #12 because it's more convenient to take advantage of the complex stack, but it's really not necessary since the module is a real number, so the "real" addition would suffice.

Here's an alternative routine *using the trace expression*; shorter but probably slower, and besides it requires two scratch matrices for the matrix product.

```
 1  LBL "FRB2"
 2  ASTO 01
 3  DIM?          ; m x n
 4  INT           ; m
 5  STO 00
 6  >",*"
 7  MAT=          ; scratch #1
 8  "*"
 9  XROM "CGTRP"
10  DIM?          ; n x m
11  FRC           ; 0,00n
12  RCL 00        ; m
13  +             ; m,00n
14  "**"          ; scratch #2
15  MATDIM
16  CLA
17  ARCL 01       ; MNAME
18  >",*,**"      ; "A,*,**"
19  M*M           ; does the work
20  MNAME?        ; MNAME
21  CMTRC         ; trace of AA*
22  SQRT
23  PURFL         ; clean house
24  "*"
25  PURFL
26  CLA
27  ARCL 01       ; MNAME
28  END
```

Row & Column Swapping and Equal Matrix Test.

A few more routines complete the utility set for complex matrices, more or less bringing it to parity with the real case. They are the column & Row swapping and the Matrix equal test.

Like **C<>C** and **R<>R** in the SandMatrix, you need to write the matrix name in ALPHA and the "*from,to*" format for the columns or rows to swap, then call the appropriate routine **ZC<>C** or **ZR<>R**. They'll exchange the selected targets in the complex matrix, which in reality will be making a double-swap if you regard this as a real matrix with double number of columns and rows.

These utilities are pivotal in the Matrix Mirror image and 90 degrees rotation routines covered in the next section. In fact they're all in the same FOCAL group; therefore the program listing is deferred.

The other routine included here is **M=M?**, a simple matrix equal test implemented as an element-based sequential comparison. Nothing earth-shattering but it comes handy when you need to do a quick check on to matrices to verify results after some operations – say to check if a matrix rotation matches the expected result.

Obviously, this routine works for real and complex matrices indistinctly, given that it operates at the element level. As usual, the routine uses **ASWAP** profusely to move the focus between the two matrices. It also sets user flag 4 if the result is positive and clears it otherwise – so you can branch programmatically in your own routines.

Program Listing:

```
01 LBL "M=M?"   ; "A,B"
02 DIM?         ; dim A
03 ASWAP        ; swap names
04 DIM?         ; dim B
05 ASWAP        ; swap back
06 X#Y?         ; different?
07 GTO 01       ; yes, no cigar
08 ,            ; pt = 1,001
09 MSIJA        ; sets A-pointer
10 *LBL 00      ; mainloop
11 MRIJA        ; A-pointer
12 MRR+         ; aij to X, incr,A-pt
13 X<>Y         ; A-pointer
14 ASWAP        ; swap names
15 MSIJA        ; sets B-pointer
16 X<>Y         ; aij to X
17 MRR+         ; bij, increases B-pt
18 ASWAP        ; swap back
19 X#Y?         ; different?
20 GTO 01       ; yep, no cigar
21 FC? 10       ; out of bounds?
22 GTO 00       ; nope, do next
23 SF 04        ; yes, it's equal!
24 "YES"
25 AVIEW        ; visual feedback
26 RTN          ; done
27 *LBL 01
28 CF 04        ; flags failure
29 "NO"         ; visual feedback
30 AVIEW
31 END          ; done
```

Eigenvalues Revisited.

The module includes three short routines that deal with finding the Eigenvalues of complex matrices.

- The first one, **ZEV22**, is for the simplest case, i.e. the 2x2 dimension matrix – where the results are obtained simply by solving a quadratic equation with the matrix trace and determinant with coefficients,λ^2 – λ.tr(A) + det(A) = 0derived from the two relationships below:

$$\mathrm{tr}(A) = \sum_{i=1}^{n} a_{ii} = \sum_{i=1}^{n} \lambda_i \qquad \det(A) = \prod_{i=1}^{n} \lambda_i$$

;

The program listing shortness is a tribute to the 41Z function set:

```
01  LBL "ZEV22"
02  XROM "CMDET"
03  ZENTER^
04  XROM "CMTRC"
05  ZNEG
06  E
07  ZREAL^
08  Z<>ST 03
09  ZQRT
10  ZAVIEW
11  MNAME?
12  END
```

- The second one, **ZEV33** is for the case of a 3x3 matrix. Here too we'll use the 41Z functions to solve the characteristic polynomial, a cubic equation with the following terms:

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

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

If the 2x2 case was simple this one is not far away from it – again thanks to the 41Z powerful and comprehensive function set, it's done with a minimalistic program shown below:

```
1.  LBL "ZEV33"
2.  XROM "?CM"
3.  XROM "CMDET"
4.  ZNEG
5.  ZENTER^
6.  XROM "CMTRC"
7.  ZNEG
8.  ZENTER^
9.  Z^2
10. ZENTER^
11. ASTO 06
12. >",#"
13. MAT=
14. MNAME?
15. M^2
16. XROM "CMTRC"
17. Z-
18. 2
19. ST/ Z
20. /
21. ZENTER^
22. 0
23. 1
24. Z<>ST 4
25. ZCRT
26. CLA
27. ARCL 06
28. END
```

The three roots will be placed in the complex stack. The first one will be displayed, and you can use **ZRDN** twice to see the other two.

- The third one, **ZEIGEN**, is an iterative approach that uses the general-purpose **ZSOLVE** routine in the 41Z module. It is therefore rather slow and very dependent on the initial guesses supplied – but on the other hand is valid for any matrix dimension. Only one Eigenvalue is found per each pair of initial guesses, so you'll need to repeat the execution to obtain the others.

  Let [I] be the identity matrix of the same dimension as [A]. The equation to solve is:

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

The program listing is also deceptively short. Note the subroutine LBL 05 used to perform the element product λI on the diagonal only – faster than using the general-purpose **ZM*** routine.

```
01 LBL "ZEIGEN"
02 ASTO 05
03 DIM?
04 "I"
05 MATDIM
06 "Z1=?"
07 PROMPT
08 ZENTER^
09 "Z2=?"
10 PROMPT
11 "*ZEV"
12 ZSOLVE
13 ZAVIEW
14 RTN
15 LBL "*ZEV"
16 "I"
17 MIDN
18 XEQ 05
19 "I,"
20 ARCL 05
21 ">",I"
22 MAT-
23 MNAME?
24 XROM "CMDET"
25 RTN
26 *LBL 05
27 DIM?
28 2
29 /
30 E3/E+
31 STO 03
32 RDN
33 *LBL 00
34 RCL 03
35 INT
36 ENTER^
37 I<>J
38 +
39 CMS
40 ISG 03
41 GTO 00
42 END
```

R00–R05 are used by **ZEIGEN** / **CMDET**, and **ZSOLVE** is data register hungry, needing R08–R12.

Example1:

Find the three Eigenvalues of the matrix given below:

$$\begin{pmatrix} 0 & 1-i & 0 \\ 1+i & 0 & 1-i \\ 0 & 1+i & 0 \end{pmatrix}$$

Tricky enough they happen to be the real values 0, 2, and -2, but assuming we don't know that beforehand let's use the guess valuesaround{ 1+i , 1-i } and { -1, 1}.

Using **ZEV33** is as simple as entering the matrix name in ALPHA and calling the program. The three eigenvalues are in the complex stack, as follows:

z1 = 0+J0
z2 = 2-J1.22E-6
z3 = -2.00+J1.22E-6 - unfortunately not a brilliant precision ...

With **ZEIGEN** it's a bit trickier. Notice how the successive values of det(A-λI) are being displayed, and how they should be converging to zero in a few iterations. Remember to adjust your display settings to reach the desired accuracy.

Example 2.-

Obtain one Eigenvalue for our example matrix C3X3, also starting with guesses { 1+i , 1-i }

or in FIX6: 2.500538 - j 1.063949

To verify this, we can use **ZEV33** again, which yields the following results:

z1 = -2.540+J15.790
z2 = 2.501-J1.064
z3 = 6.040+J3.274

PS. This web page is a very good resource to check your results:
https://www.arndt-bruenner.de/mathe/scripts/engl_eigenwert2.htm

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

calculate

```
Characteristic Polynomial:
  -x^3 + 6x^2 + 38x - 75 + (18x^2 - 131x + 289)·î

Complex Eigenvalues: { -2.5402328323210113+15.790341733496 31î ;  2.5005380822

Eigenvectors:

 for Eigenvalue -2.5402328323210113+15.790341733496 31î:
   [ -0.5827456938021328-0.7396115289132087î ; 0.973154380482974+0.5153261763

 for Eigenvalue 2.5005380822524814-1.0639492474549757î:
   [ 0.3668388788943956+0.6212757392135828î ; -0.346778932445011 8-0.646230670

 for Eigenvalue 6.03969475006853+3.2736075139586642î:
   [ -1.1576870840436446+0.6006681697441119î ; -1.536194323157123-1.985269639
```

The path not taken: using diagonal & cross-elements product sums.

Another alternative for **ZEV33** that does not require calculating the square matrix is described before. The advantage is that it doesn't need the additional memory for the scratch matrix needed in the square matrix operation, but it's a longer program and it takes much longer to complete.

Characteristic polynomial equation:

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

The calculation for c2 is done as ΣZIJJI – ZMDPS, with the routines listed below:

1	LBL "ΣZIJJI"	
2	XROM "?CM"	
3	E	
4	E3/E+	
5	ZIJ=	
6	CLSTZ	
7	LBL 01	
8	ZIJ?	i,00j
9	INT	i
10	I<>J	0,00i
11	LASTX	i
12	+	i,00i
13	ZIJ=	
14	ZJ+	i,00(i+1)
15	FS? 10	out of bounds?
16	ZRDN	
17	FS? 10	
18	RTN	
19	LBL 00	
20	ZIJ?	r,00s
21	I<>J	s,00r
22	ZIJ=	
23	XROM "CMR"	
24	ZENTER^	
25	ZIJ?	
26	I<>J	s,00r
27	ZIJ=	
28	VIEW X	
29	XROM "CMR"	
30	Z*	
31	Z+	
32	ZENTER^	
33	ZJ+	
34	FC? 09	
35	GTO 00	
36	GTO 01	
37	END	

1	LBL "ZMDPS"		
2	CLSTZ		
3	CF 00		
4	DIM?		
5	6		
6	X<=Y?	6<dim ?	
7	SF 00	3x3 or hgher	
8	E		
9	E3/E+	1,001	
10	ZIJ=		
11	LBL 00		
12	ZIJ?		
13	XROM "CMR"	aij	
14	ZENTER^		
15	ZJ+	next col	
16	FS? 09		
17	GTO 01		
18	ZI+	next row	
19	ZIJ?		
20	XROM "CMR"		
21	Z*		
22	Z+		
23	ZENTER^		
24	FC? 10		
25	GTO 00		
26	LBL 01		
27	FC? 00		
28	ZRDN	undo last push	
29	FC? 00		
30	GTO 02		
31	E		
32	E3/E+		
33	ZIJ=		
34	XROM "CMR"		
35	LBL 02		
36	ZRDN		
37	END		

Characteristic Polynomial, Determinant & Inverse

This section finally tackles the general-purpose case for the three main subjects covered previously in the manual: Eigenvalues, Determinant and Inverse of a square matrix.

As we know, the Eigenvalues are calculated as the roots of the characteristic polynomial. In the 41Z module there are programs to calculate the roots of complex polynomials of any degree, therefore we'll address the calculation of the coefficients of the characteristic polynomial here, using the Faddevv-Leverrier method as we did in the SandMatrix for the case of real matrices

The beauty of this approach is that we'll put those coefficients to work bigtime to obtain the matrix determinant (independent tern), and also the matrix inverse using the corollary of the Cayley-Hamilton theorem. So, we'll kill three birds with a single stone (well, maybe with a single stone plus one extra pebble ;-)

1. **ZPCHR** calculates the coefficients of the characteristic polynomial of the square matrix whose name is in ALPHA. The program will enumerate them and will return the control word *bbb.eee* in X, denoting the **complex** registers holding the coefficients. The original matrix is not altered, and its name is left in ALPHA upon completion.

2. **CMDET** (in the SandMatrix) also uses this approach, whereby the determinant is the last coefficient of said characteristic polynomial with changed sign:

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

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

3. Lastly, **ZMINV** calculates the inverse matrix (*), using the expression:

   $$A^{-1} = \frac{-1}{c_n} \left(A^{n-1} + c_1\, A^{n-2} + \cdots + c_{n-2} A + c_{n-1}\, I \right).$$

   Note that this expression can be re-written using the Honer form,

   $$A^{-1} = \frac{-1}{c_n} \left(c_{n-1}\, I + A(c_{n-2} I + A(c_{n-3} I + \cdots + A(c_1 I + A\, c_0 \right)$$

   Which is much more convenient to use the powerful MCODE matrix functions available in the SandMatrix, **MAT+** and **M*M**, plus the routine **Z*M** needed to do the final bit, i.e. the division by det(A).

   Note that *the program does not replace the original matrix with its inverse*, instead the inverse matrix is left in X-Mem with the name "P".

(*) Remember that the MCODE function **MINV** does support complex matrices, and therefore will always be the preferred method for the matrix inversion calculation- however **ZMINV** is added to the module for comparison purposes.

References:

https://en.wikipedia.org/wiki/Faddeev%E2%80%93LeVerrier_algorithm

https://en.wikipedia.org/wiki/Cayley%E2%80%93Hamilton_theorem

Example.- Calculate the characteristic polynomial, determinant and inverse for the C3x3 matrix:

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

First off, ensure the matrix is loaded using XEQ **"^C33**" if still not in X-Mem.

Then with the matrix name in ALPHA we do:

"C3X3", XEQ "ZPCHR", => shows iterations and then:

C0 = 1+J0 R/S
c1 = -6-J18 R/S
c2 = -38+J131 R/S
c3 = -75+J289 thus: det = 75-J289

And now:

XEQ "ZMINV" (shows counters…) SF 21, XEQ "ZOMR"

1,1= 0.1241+J0.0647 1,2=-0.0152+J0.0213 1,3= 0.0104+J0.1599
2,1= 0.0213-J0.1578 2,2= 0.1513-J0.0704 2,3=-0.0322-J0.1508
3,1= 0.2546+J0.0343 3,2=-0.0004+J0.1316 3,3= 0.1813+J0.0187

In order to test the accuracy, let's invert the inverse to compare it with the original – calculating the norm of the difference:

ZMNRM [C3X3 – Inv(Inv(C3X3))] = 2.2289235 E-08

The accuracy loss happens due to the FOCAL subroutines called by the program, despite the other MCODE functions used. Still it's a pretty decent performance considering the inherent platform limitations and the nature of the iterative process.

Register, Flags, etc.

Data registers {R00 – R03} are used in main memory.

Only user flag 4 is used to differentiate between the main routines.

Two auxiliary matrices are needed, each of the same size of the original one – therefore the maximum complex matrix size is n= 14, for a total of 588 X-mem registers. This requirement 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!

LBL "Z*ID" - This subroutine does the equivalent to **MAT*** in the case "X,I,I", with a complex value in {Y,X} and the identity matrix [I]. This is faster than using **Z*M**, since only the diagonal elements need doing:

$$z.[\mathrm{I}] = \begin{bmatrix} z & (0) \\ (0) & z \end{bmatrix}$$

Program Listing.

```
*LBL "ZMINV"
SF 04
GTO 04
*LBL "ZPCHR"
CF 04
*LBL 04
XROM "?CM"
ASTO 00
 >",P"
MAT=
DIM?
"#"
MATDIM          ; 2n,002n
FRC             ; 0,002n
2
/               ; n,00n
LASTX
+               ; 2,00n
STO 01          ; counter
CF 21
*LBL 00         ; loop1
VIEW 01
XEQ 01          ; Ck
XROM "Z*ID"     ; [#]=Ck*[#]
"P,#,#"
MAT+
CLA
ARCL 00
>",#,P"
M*M
ISG 01          ; next?
GTO 00          ; yes
XEQ 01
FS? 04          ; inverse?
GTO 04          ; yes, ->
 E
STO 02
CLX
STO 03          ; c0 = 1
RCL 01
E3/E+
SF 21
ZOUPT  (ZF# 52)
CLA
ARCL 00
PDEG            (ΣF# 93)
INT
ENTER^
 ST+  X
 2
 +
 E6
 /
 6
 E3/E+
 +
 REGMOVE
 X<>Y
 XROM "ZPROOT"
 RTN
LBL "Z*ID"      ; z*[I]
"#"
MIDN
1.001
XROM "CMS"
 >","
DIM?
FRC
3,00002         ; counter
+               ; 3.00m02
1.001
ENTER^
ENTER^
ST+ Y           ; 2.002
*LBL 02         ; loop2
 X<>Y
ST+ Z           ; k.00k
X<>Y            ; 2.002
MMOVE           ; 1.001
 ISG T
 GTO 02
 RTN
 *LBL 01
 "P"
 CMTRC
 RCL 01
 INT
  E
 -
 CHS
 ST/  Z
 /
  ZSTO IND 01    (129)
 RTN
*LBL 04         ; inverse
CLA
ARCL 00
>",P"
MAT=
RCL 01          ; n+1,00n
FRC
 E-3            ; 0,001
-
2
+               ; 2,00(n-1)
STO 01
PDEG            (ΣF# 93)
X<0?
GTO 04
*LBL 05         ; loop3
ZRCL IND 01     (129)
XROM "Z*ID"     ; [#]=Ck*[#]
"P,#,#"
MAT+
CLA
ARCL 00
>",#,P"
M*M
ISG 01          ; next?
GTO 05          ; yes, ->
LBL 04
ZRCL IND 01      (129)
XROM "Z*ID"     ; [#]=Cn*[#]
"P,#,P"
MAT+
RCL 01          ; n,00(n-1)
 E
+               ; n+1
ZRCL IND X      (243)
ZNEG
ZINV
"#"
PURFL
MNAME?
 XROM "Z*M"
ZOMR
END
```

Real Matrix version of the same programs.

It's easier to handle real matrices with the SandMatrix functions – especially **MAT*** which was the trouble child in the complex case. I've used the same subroutines *LBL 01 and *LBL 03 (for "**Z*ID**), so it's easier to establish the comparisons between both cases. Note that this version does not calculate the Eigenvalues – you can use **PROOT** for that purpose.

So what's the additional benefit of these over **MDET** and **MINV** ? Nothing much to be honest, supposedly the accuracy should hold better using the Faddeev-Leverrier algorithm but the difficult examples it was tried on proved otherwise.

```
 1 *LBL "DET"
 2 CF 04
 3 GTO 04
 4 *LBL "INV"
 5 SF 04
 6 *LBL 04
 7 ASTO 00
 8 >",P"
 9 MAT=
10 DIM?
11 "#"
12 MATDIM
13 FRC
14 2
15 +
16 STO 01
17 CF 21
18 *LBL 00
19 VIEW 01
20 XEQ 01          ; nth. Coef.
21 XEQ 03          ; x*[#]
22 "P,#,#"
23 MAT+
24 CLA
25 ARCL 00
26 >",#,P"
27 M*M
28 ISG 01
29 GTO 00
30 XEQ 01
31 CLD
32 CLA
33 ARCL 00
34 FS? 04
35 GTO 04
36 E
37 X<> 01          ; n+1
38 CHSYX
39 CHS             ; (-1)^n
40 RTN
41 *LBL 01         ; n-th coef.
42 "P"
43 MTRACE
44 RCL 01
45 INT
46 E
47 -
48 /
49 CHS
50 STO IND 01
51 RTN
52 *LBL 03         ; x*[#]
53 "#"
54 MIDN
55 "X,#,#"
56 MAT*
57 RTN
58 *LBL 04         ; inverse
59 >",P"
60 MAT=
61 RCL 01
62 FRC
63 E-3
64 -
65 2
66 +
67 STO 01
68 PDEG
69 X<0?
70 GTO 04
71 *LBL 05
72 RCL IND 01      ; Cn
73 XEQ 03          ; x*[#]
74 "P,#,#"
75 MAT+
76 CLA
77 ARCL 00
78 >",#,P"
79 M*M
80 ISG 01
81 GTO 05
82 *LBL 04
83 RCL IND 01
84 XEQ 03
85 "P,#,P"
86 MAT+
87 RCL 01
88 E
89 +               ; (n+1)
90 RCL IND X       ; Cn
91 CHS             ; -Cn
92 1/X             ; -1/Cn
93 "X,P,P"
94 MAT*
95 MNAME?          ; "P"
96 SF 21
97 OMR
98 END
```

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

Element Rotations – Matrix Mappings.

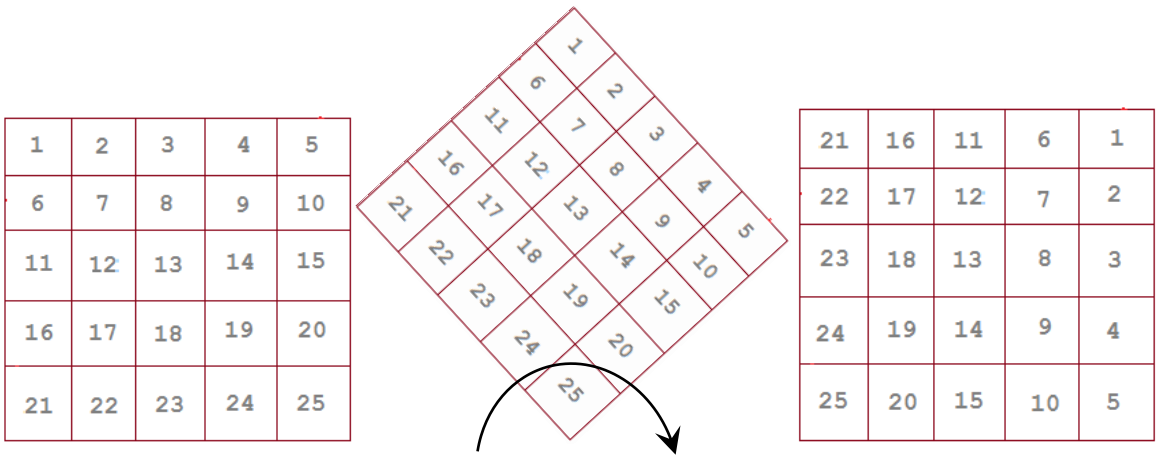
The concept of in-matrix mapping refers to the different operations made to the elements of the matrix according to given rules. One such mapping very commonly used is the matrix transposition, whereby each element aij is swapped with its "counterpart" one, aji. Another type of mapping deals with element "rotations", where there's a shifting or rotation of all elements by one of several positions (steps) in either clockwise (right) or counter-clockwise (left) directions. Let's see them individually.

Matrix 90-degree Rotations. Mirror Images.

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

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

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



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

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

For real matrices it's an in-place algorithm by virtue of the **TRNPS** MCODE function, but for complex matrices such is not the case, as using the **CMTRP** FOCAL routine the transposed coexists in memory with the original during the operation.

Program usage:

Just type the matrix name in ALPHA and execute **ZM90R** or **ZM90L** depending on the desired direction of rotation. For **ZMIRR** you need to clear or set user flag manually to indicate vertical or horizontal mirror image respectively. Note that the utilities **ZC<>C** and **ZR<>R** do not use the user flag, as they can be also called from other programs/

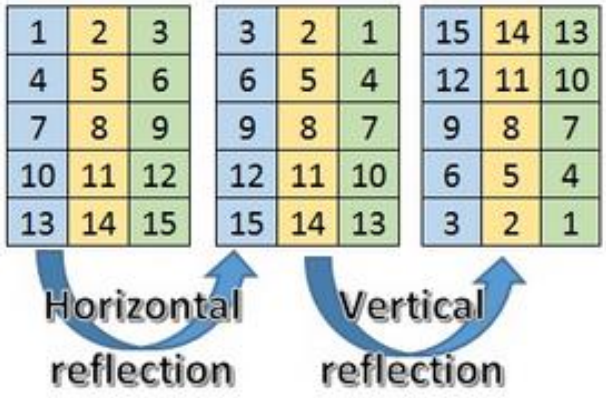
Program Listing: (includes ZC<>C and ZR<>R)

```
01  LBL "ZM90R"     ; right
02  CF 00           ; flag case
03  GTO 00          ; merge
04  LBL "ZM90L"     ; left
05  SF 00           ; flag case
06  *LBL 00         ; common
07  XROM "CMTRP"    ; transpose
08  LBL "ZMIRR"     ; mirror
09  DIM?            ;2n x 2m
10  2
11  /               ; n x m
12  FRC             ; 0,00m
13  E
14  +               ; 1,00m
15  *LBL 01
16  FC? 00          ; right?
17  XROM "ZC<>C"
18  FS? 00          ; left?
19  XROM "ZR<>R"
20  E-3             ; next col?
21  -
22  ISG X           ; next row
23  GTO 01          ; repeat
24  RTN             ; done.
25  LBL "ZC<>C"     ; iii,jjj
26  ST+ X           ; realpt
27  C<>C            ; swap
28  E               ; next col
29  E3/E+           ; next col
30  +
31  C<>C            ; swap
32  GTO 00          ; merge
33  LBL "ZR<>R"     ; iii,jjj
34  ST+ X           ; real pt.
35  R<>R            ; swap
36  E               ; next row
37  E3/E+           ; next row
38  +
39  R<>R            ; swap
40  *LBL 00         ; common
41  LASTX           ; 1,001
42  -               ; undo
43  2               ; cpx pt.
44  /
45  END             ; done
```

With this one already safely under our belt let's move on to the next section that deals with Layer Rotations in steps of a single element.

Terminology alert:

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



Matrix Layer Rotations.

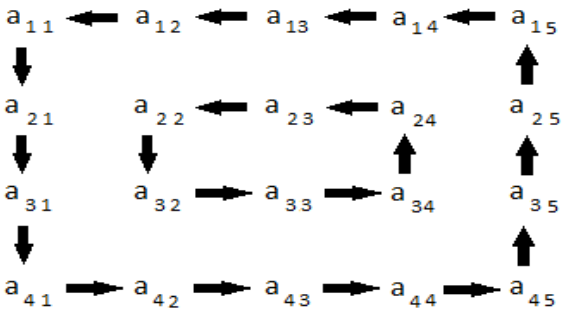
Rotating the matrix elements by one step needs to be done for all layers (or rings) in the matrix, from the outermost to the innermost – which depending on the dimensions will remain unmodified when the number of rows is odd.

For example, the counter-clockwise rotation of a 4 x 5 matrix is represented by the figure below. Note that in one rotation, you only have to shift elements by one "step".

For a counter-clockwise (left) rotation we need to shift each layer elements as follows:

1. one position to the left within the upper part
2. one position downwards within the left part
3. one position to the right within the bottom par
4. one position upwards within the right part

Then such should be repeated for all inner layers, until completing the whole matrix.



A clockwise rotation follows the same process with the shifting done in opposite directions as stated above.

Let's denote the original (m x n) matrix as [A] = (ai,j), and the rotated matrix [B] = (bij), also m x n.The following general expressions describe the algorithm required to perform **one single step, clockwise** rotation:

```
FOR k = 0 TO int[min(m,n)/2]          ; current layer

   FOR j = k+1 TO (n-1-k)             ; top row rightwards
      b(1+k),(j+1) = a(1+k),j
   NEXT j

   FOR i = (1+k) TO (m-1-k)           ; rightmost column downwards
      b(i+1),(n-k) = a i,(n-k)
   NEXT i

   FOR j = (n-k) TO (2+k) STEP -1     ; bottom row leftwards
      b(m-k),(j-1) = a(m-k),j
   NEXT j

   FOR i = (m-k) TO (2+k) STEP -1     ;leftmost column upwards
      b(i-1),(1+k) = a i,(1+k)
   NEXT i

NEXT k                                ; next layer
```

All we need to do is figure out a way to implement this simple BASIC-like approach as an RPN routine, which as everybody knows is far from being the RPN's forte.

The routines included require an auxiliary matrix to hold the rotated elements, therefore they don't offer an in-place solution to the problem. Solutions for both real and complex matrices are provided.

Note as well that to do a **counter-clockwise** rotation we can apply the *same algorithm on the transposed matrix*, therefore we only need to implement the clockwise case.

Copying and Swapping Elements

These ad-hoc utilities should facilitate the task, copying elements from a source matrix to a destination one, using their pointers in the stack registers X (destination) and Y (source). As always, source and destination matrix names are expected in ALPHA separated by comma.

```
01. LBL "a<>b"
02. X<>Y
03. MSIJA
04. MR
05. X<> Z
06. ASWAP
07. MSIJA
08. MR
09. X<> Z
10. MS
11. R^
12. ASWAP
13. MSIJA
14. R^
15. MS
16. X<> T
17. RTN
18. LBL "a>b"
19. X<>Y
20. MSIJA
21. MR
22. RCL Z
23. ASWAP
24. MSIJA
25. X<> Z
26. MS
27. ASWAP
28. RDN
29. RTN
30. LBL "Ca>b"
31. X<>Y
32. CMR
33. ASWAP
34. RCL Z
35. CMS
36. ASWAP
37. END
```

All in all, nothing more than the usual **ASWAP** dance to set the focus on the target matrix, spiked out with calls to **MR** and **MS** here and there to move the elements between the matrices. Notably the complex case **Ca>b** turns out to be shorter due to the help from the **CMR** and **CMS** utilities of course.

Discarding trivial cases.

The other auxiliary routine is **?1x1**, used to discard the trivial case where there's only one column or one row in the matrix. This routine uses an input parameter in the X-register (a.k.a the "threshold"), that must be equal to 1 for real matrices or 2 for complex ones.

```
01. LBL "?1X1"
02. DIM?
03. X<>Y          ; eiether 1/2
04. E3/E+
05. -
06. INT           ; rows
07. X=0?
08. DIMERR
09. LASTX
10. FRC           ; cols
11. X=0?
12. DIMERR
13. END
```

Equipped with our shiny new utilities, here's the complete Matrix Layer Clockwise Rotation routines for the real and complex cases combined into a single program.-

Program listing:

```
1   LBL "M<)R"
2   CF 02          ; flag case
3   E              ; single
4   GTO 10         ; merge
5   LBL "ZM<)R"
6   XROM "?CM"     ; complex?
7   2              ; dual
8   SF 02          ; flag case
9   *LBL 10        ; common
10  XROM "?1X1"    ; threshold?
11  >",#"          ; scratch
12  MAT=           ; copy over
13  DIM?           ; 2m x 2n
14  STO 04
15  2
16  /              ; m x n
17  FS? 02         ; complex?
18  STO 04
19  RCL 04         ; m x n
20  2
21  /              ; (m/2),00(n/2)
22  INT            ; m/2
23  E
24  -
25  I<>J           ; 0,00(1-m/2)
26  STO 00         ; range for k
27  *LBL 00        ; current layer
28  RCL 04
29  FRC            ; 0,00n
30  RCL 00
31  INT            ; k
32  E
33  +              ; 1+k
34  STO 02
35  I<>J           ; 0,00(1+k)
36  -
37  RCL 02         ; 1+k
38  +
39  STO 01         ; (k+1),00(n-k-1)
40  *LBL 01        ; current row
41  E              ; increase column
42  XEQ 05         ; shift row
43  ISG 01         ; NEXT i
44  GTO 01         ; do next row
45  RCL 04
46  INT            ; m
47  RCL 02         ; k+1
48  -
49  I<>J           ; 0,00(m-k-1)
50  RCL 02         ; 1+k
51  +
52  STO 01         ; (k+1),00(m-k-1)
53  RCL 04
54  RCL 00
55  I<>J           ; x,00k
56  -
57  FRC            ; 0,00(n-k)
58  STO 02
59  *LBL 02        ; current column
60  E              ; increase row
61  XEQ 06         ; shift column
62  ISG 01         ; next J
63  GTO 02         ; do next column
64  RCL 04
65  FRC            ; 0,00n
66  I<>J           ; n
67  RCL 00
68  INT            ; k
69  -
70  E
71  LASTX          ; k
72  +              ; 2+k-1
73  I<>J           ; 0,00(2-k-1)
74  STO 03
75  +              ;(n-k),00(2-k-1)
76  STO 01
77  RCL 04
78  RCL 00         ; k,00s
79  -
80  INT            ; m-k
81  STO 02
82  *LBL 03        ; current row
83  -E             ; decrease column
84  XEQ 05         ; shift row
85  DSE 01         ; NEXT i
86  GTO 03         ; do next row
87  RCL 02         ; m-k
88  RCL 03         ; 0,00(2-k-1)
89  +
90  STO 01         ; ;(m-k),00(2+k-1)
91  RCL 00
92  INT            ; k
93  E
94  +              ; 1+k
95  I<>J           ; 0,00(1+k)
96  STO 00
97  *LBL 04        ; current row
98  -E             ; decrease row
99  XEQ 06         ; shift column
100 DSE 01         ; NEXT j
101 GTO 04
102 ISG 00         ; NEXT k
103 GTO 00         ; do next layer
104 ASWAP
105 MAT=           ; final copy
106 PURFL          ; kill scratch
```

```
107  MNAME?          ; recall name
108  RTN             ; all done
109  *LBL 05         ; shift row
110  I<>J
111  RCL 01
112  INT
113  I<>J
114  GTO 07          ; merge
115  *LBL 06         ; shift column
116  RCL 01
117  INT             ; row pointer
118  *LBL 07         ; common
119  RCL 02          ; cpx. pointer
120  +               ; row,col
121  +               ; adds offset
122  LASTX           ; source pt.
123  X<>Y            ; destination Pt.
124  FC? 02          ; real?
125  XROM "a>b"
126  FS? 02          ; complex?
127  XROM "Ca>b"
128  END
```

Program remarks:

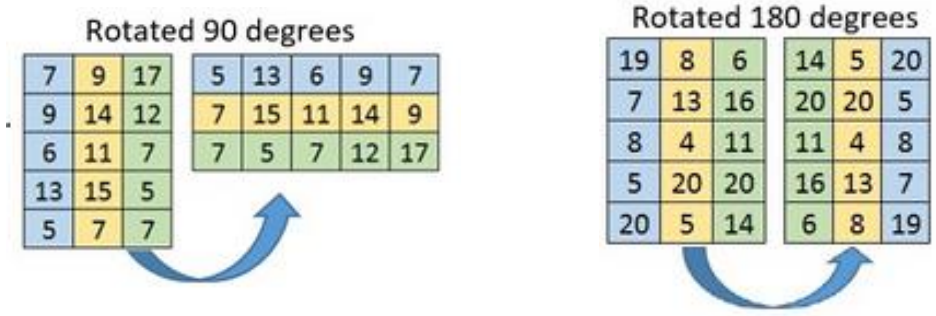
The program uses user flag 02 and data registers R00 – R04, plus a scratch matrix "**#**" of the same size as the original. The data registers hold the following parameters:

R00 – layer index
R01 – row or column index
R02 – scratch
R03 – scratch
R04 – matrix dimension

The routine is not difficult in nature but prepping the ranges for all loops is sticky and error-prone. Especially tricky is the descending counter in the DSE loops, which requires subtracting one to the final value to be equivalent to the FOR/NEXT structure *with negative steps*.

The program structure reflects the general algorithm, so there are four small loops (one for each part of the layer) ran sequentially inside one larger one that does the layers. The small loops are LBL 01, LBL 02, LBL 03 and LBL 04, inscribed in rectangles on the listing above.

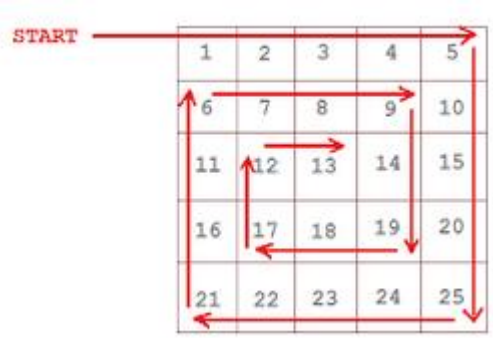
Two routines - LBL 05 and LBL 06 - do the element shifting on the rows (upper and bottom parts of layer) and columns (right and left parts of layers) respectively. Each one is called multiple times within the loops (as many as elements need shifting), and the input parameters indicate whether it is an advance or a back-step (positive or negative sign).



Corollary: Matrix Spiral Input/Output. { **ZIMS** , **ZOMS** }

Adding a matrix "spiral" input/output capability is not difficult with just a few modifications to the previous program. The idea is to navigate the source matrix following the same algorithm, simply showing the current element values and offering changing in the input case. Obviously we won't be needing an auxiliary matrix for this task, thus the parts of the code dealing with that will be bypassed – controlled by user flag 01.

Like we did in the Input/Output routines by rows and columns, user flag 00 will discriminate between the input (set) and output (clear) cases. User flag 02 will continue to flag the real vs. Complex cases as before, but we're going to need two more flags to deal with the possible "left-over "elements, i.e. those not belonging to any rotation layer.



In the example below there's indeed a left-over element (a32= 13) but in general there may be more than one of course, all depending on the matrix dimensions.

Left-Over elements.

The first thing is to check if there are left-over elements. This is a consequence of the matrix dimensions, where the number of layers is: L = int [min(m,n) / 2]. As it happens, if **min(m,n)** is an odd number then we'll have left-over elements. Moreover, these will be arranged in row order or in column order, depending on which one was the minimum. Therefore, we'll need two user flags to mark the possible conditions, as described by the table below:

User Flag	**F 00**	**F 01**	**F 02**	**F 03**	**F 04**
Clear	Output	Rotation	Real	Whole Layers	Column order
Set	Input	Spiral Print	Matrix	Left-over elms.	Row order

The last problem to tackle is determining *how many* left-over elements exist in the orphan row or column. This is also determined by the matrix dimensions of course, and the implementation uses a final loop using the following ranges:

Let r = min (m,n); L = number of layers; i/j = leftover row / column.

```
FOR h = (L+1) TO (r-L)
   Show element ai,h or ah,j depending on F04
NEXT h
```

Four global labels are provided to do the spiral printing: two for Input / Output plus **IMS** and **OMS** - another two for Real / Complex cases **ZIMS** and **ZOMS**. One additional global label is included **"**>**ZIJ**" to display the complex elements - which is shared with the "classic" matrix Input / Output by columns and rows, { **ZIMC**, **ZOMC**, **ZIMR**, and **ZOMR}**. That's quite a set!

Bearing all the above under consideration it results into a non-trivial modification of the rotations program, refer to the program listing in next page for the final implementation details.

Program listing:

```
*LBL "IMS"
3
GTO 11
*LBL "OMS"
2
GTO 11
*LBL "ZIMS"
7
GTO 11
*LBL "ZOMS"
6
*LBL 11
X<>F
FS? 02
XROM "?CM"
GTO 11
*LBL "ZM<)R"
XROM "?CM"
2
ENTER^
4
GTO 10
*LBL "M<)R"
E
0
*LBL 10
X<>F
RDN
XROM "?1X1"
"`,#"
MAT=
*LBL 11  ; common
DIM?
STO 04
2
/
 FS? 02
 STO 04
 RCL 04
 INT
 LASTX
FRC
I<>J
X<Y?
SF 04    ; mark #1
X>Y?
X<>Y
ODD?
SF 03     ; mark #2
 2
 /
  INT
  E
  -
I<>J
  STO 00
*LBL 00
  RCL 04
FRC
  RCL 00
  INT
E
+
  STO 02
I<>J
  -
  RCL 02
  +
  STO 01
*LBL 01
E
XEQ 05
  ISG 01
GTO 01
  RCL 04
  INT
RCL 02
  -
I<>J
  RCL 02
  +
  STO 01
  RCL 04
  RCL 00
  I<>J
  -
  FRC
  STO 02
*LBL 02
  E
  XEQ 06
  ISG 01
GTO 02
RCL 04
FRC
I<>J
RCL 00
 INT
 -
 E
LASTX
 +
I<>J
 STO 03
 +
 STO 01
RCL 04
 RCL 00
 -
 INT
 STO 02
*LBL 03
 - E
XEQ 05
DSE 01
GTO 03
 RCL 02
 RCL 03
 +
 STO 01
RCL 00
 INT
  E
 +
 I<>J
 STO 02
*LBL 04
 - E
 XEQ 06
 DSE 01
GTO 04
 ISG 00
GTO 00
 FS? 01
GTO 08
ASWAP
MAT=
 PURFL
MNAME?
 RTN
*LBL 08
 FC? 03
 RTN
 RCL 04
FS? 04
I<>J
FRC
RCL 00
 INT
 STO 00
E
 +
 +
 RCL 00
I<>J
```

```
156    -
157    STO 00
158    FC? 02
159    GTO 08
160    ZJ+
161    *LBL 09
162    ZIJ?
163    >ZIJ
164    FC? 04
165    ZJ+
166    FS? 04
167    ZI+
168    ISG 00
169    GTO 09
170    RTN
171    *LBL 08
172    J+
173    *LBL 12
174    XEQ 13
175    FC? 04
176    J+
177    FS? 04
178    I+
179    ISG 00
180    GTO 12
181    RTN
182    *LBL 13
183    "a"
184    MRIJ
185    ΣV#
186    64          ; >aIJ
187    AVIEW
188    MNAME?
189    FS? 00
190    STOP
191    MS
192    RTN
193    *LBL 05
194    I<>J
195    RCL 01
196    INT
197    I<>J
198    GTO 07
199    *LBL 06
200    RCL 01
201    INT
202    *LBL 07
203    RCL 02
204    +
205    +
206    LASTX
207    FS? 01
208    GTO 07
209    X<>Y
210    FC? 02
211    XROM "a>b"
212    FS? 02
213    XROM "Ca>b"
214    RTN
215    *LBL 07
216    FC? 02
217    MSIJ
218    FC? 02
219    GTO 13
220    >ZIJ
221    END
```

The central part of the program is practically unchanged from how it was in the matrix rotations. The new stuff starts after all rotation layers are done, see steps 134 to 195. The section dealing with left-over elements starts at step 142, see LBL 09 and LBL 12 for the complex and real cases respectively. Also notice how some of the routines have been modified, like LBL 07 starting at step 205.

There you have it, what started being a curiosity has evolved into a relatively complex program… certainly it wasn't anticipated from the initial design.

Matrix Layer Rotations – Alternative Approach.

Just in case you thought we were done with the rotation section, here's a follow-up that uses an alternative approach with data registers instead. The core routines are contributed by Jean-Marc Baillard, and they follow a different element storage convention for the matrix, done by columns – not by rows – in sequential data registers.

I've added "driver routines to "interface" with those core routines, using "standard" matrices as starting point – and leaving the result also in standard "SandMatrix" format. This is simple for the real case, where only a transposition is required to convert between formats (remember that the SandMatrix also supports storing matrices in main memory, using the "Rxxx" convention for the matric name) – but it was quite challenging for the complex case: JM's convention places the real and real and imaginary parts also into sequential data registers, which requires intermediate utilities dealing with data registers block "splicing" and "splitting", for lack of better terms. Let's see them next…

Program	Real	Inputs	Complex	Inputs
Driver	**MROT+**	Matrix Name	**ZMROT+**	Matrix Name
Core	**MROT**	bbb, eee, #steps	**ZMROT**	bbb, eee, #steps

- The driver programs only do one clockwise element rotation, but transposing it *__twice__* (i.e. doing nothing!) also reverses the direction of rotation- therefore it's very easy to compare the results using **M<)R** and **ZM<)R**.

- The core routines do a counter-clockwise rotation, of any number of steps- thus multiple steps can be done with just one execution of the routine, as opposed to M**<)R** and **ZM<)R** described before.

- The core routine MROT uses four input parameters: as follows:
  - T = bbb = address of the 1st coefficient > 8
  - Z = m = Nb of rows > 1
  - Y = n = Nb of columns > 1
  - X = k = Nb of rotations > 0

Real Matrices: here it's not even needed to transpose the matrices: the combined effect (by columns + counter-clockwise) = (by rows + clockwise)

```
1   LBL "MROT+"      ; MNAME
2   E                ; threshold
3   XROM "?1X1"
4   "|-,R9"          ; starts at
    R10
5   MAT=             ; copy matrix
6   E1               ; first register
7   DIM?             ; m x n
8   INT              ; m
9   LASTX
10  FRC              ; 0,00n
11  I<>J             ; n
12  E                ; one rot.
    step
13  XROM "MROT"
14  ASWAP            ; "#,MNAME"
15  MAT=
16  MNAME?
17  RTN
```

Complex Matrices: Here we'll use the same process twice, on the real and imaginary components of the given matrix - since each of those is a real matrix and therefore can be dealt with by the previous case.

1. Separate the complex matrix into real and imaginary parts, both in X-Mem
2. Copy them sequentially to main memory, starting in R10. This will use R09 for the matrix header.
3. Do the rotation on each component and copy it back to X-mem
4. Re-assemble the complex matrix from its (rotated) components.

An additional complication in the Complex case occurs due to the lack of more scratch registers to save the original matrix name. We'll circumvent this using the complex stack as an auxiliary memory area; for example the ALPHA string containing the matrix name

- To store it we use: RCL M, RCL N, **ZENTER^**
- And to restore it: **ZRDN**, STO N, X<>Y, STO M

Note that this will not work in manual more or SST'ing a program because the functions will call ZAVIEW and it can't cope with ALPHA DATA – but it's perfectly safe in running program. Nice trick in my book, not quite the original design intent but very useful indeed.

Example: let's do some rotations using our trusted example matrix C3X3. The idea is to first use **ZM<)R** and compare the result with the output of **ZMROT+**, to check if they're the same.

"C3X3,ROT1", **MAT=**, **MNAME?** , XEQ "ZM<)R", -> ROT1 is the rotated matrix
"C3X3,ROT2", **MAT=**, **MNAME?**, XEQ "ZMROT+" -> ROT2 is the rotated matrix
"ROT1,ROT2", **M=M?** -> "YES" (hopefully ;-)

Program listing:

```
01 LBL "ZMROT+"
02 2
03 XROM "?1X1"
04 >",RE,IM"
05 RCL M
06 RCL N
07 ZENTER^
08 XROM "ZM>XY"
09 "RE"
10 XEQ 00
11 "IM"
12 XEQ 00
13 CLA
14 ZRDN
15 STO N
16 X<>Y
17 STO M
18 ASWAP
19 XROM "XZ>ZM"
20 PURFL
21 ASWAP
22 PURFL
23 ASWAP
24 DIM?
25 MNAME?
26 RTN
27 *LBL 00
28 >",R9"
29 MAT=
30 E1
31 DIM?
32 INT
33 LASTX
34 FRC
35 I<>J
36 E
37 XROM "MROT"
38 ASWAP
39 MAT=
40 END
```

The path not taken - Complex Matrices: An alternative top-level process using **ZMROT**:

1. Separate the complex matrix into real and imaginary parts, both in X-Mem
2. Copy them to main memory, starting at R(m.n) for the imaginary part and R01 for the real part. Note how by copying first the imaginary parts and later the real parts we get around removing the status register placed in R(m.n) by the first copy operation – ending with a nice continuous pair of data blocks: [RE]-[IM]
3. Splice the data blocks into a common one, so the coefficients are presented in the format expected by **ZMROT**, which now can be called to do the rotation. [RE-IM]
4. Next Split the result matrix into two blocks, for the real and imaginary matrices
5. Merge them into a common matrix in X-Mem – overwriting the original.

For step #3 - storing the coefficients in sequential registers real part first and imaginary part following, poses an interesting challenge to the "interfacing" routine. For step #4after the rotation, we need to de-splice such an arrangement into two separate data register blocks, one for the real parts and another for the imaginary parts. Only once this is done we can call our trusted **XY>ZM** routine to build the complex matrix in "SandMatrix" format.

An additional complication in the Complex case occurs because **ZMROT** assumes that all data registers (starting with R01) are allocated to the matrix coefficients. This leaves us with just one scratch register in main memory, R00. This we'll circumvent using the complex stack as an auxiliary memory area; for example, the ALPHA string containing the matrix name

- To store it we use: RCL M, RCL N, **ZENTER^**
- And to restore it: **ZRDN**, STO N, X<>Y, STO M

Nice trick in my book, not quite the original design intent but very useful indeed.

Note that the implementation uses an auxiliary scratch area to hold the copied values temporarily, and that area is moved *en-masse*once the process is completed. - so it's not an in-place solution. The data blocks are each of m.n data registers, and the resulting spliced block is twice that size, i.e. 2.m.n

Routine listing:

```
01 LBL "ZMROT+"
02 2
03 XROM "?1X1"
04 ">", RE,IM"
05 RCL N
06 RCL M
07 ZRPL^
08 XROM "ZM>XY"
09 "IM,R"
10 DIM?
11 MSZE?
12 ZENTER^
13 STO  L(4)
14 AINT
15 MAT=
16 "RE,R0"
17 MAT=
18 LASTX
19 XROM "SPLICE"
20 ZRDN
21 ZENTER^
22 X<>Y
23 INT
24 LASTX
25 FRC
26 I<>J
27 E
28 XROM "ZMROT"
29 ZRDN
30 XROM "SPLIT"
31 "RE,R0"
32 DIM?
33 ASWAP
34 MATDIM
35 MAT=
36 "IM, R"
37 MSZE?
38 AINT     ; IM,Rxx
39 DIM?
40 ASWAP ; Rxx,IM
41 MATDIM
42 MAT=
43 CLA
44 ZRDN
45 X<>Y
46 STO N
47 X<>Y
48 STO M
49 ASWAP
50 XROM "XY>ZM"
51 END
```

Appendix. Matrix Scrolling by ALPHA

These routines scroll the matrix whose name is in ALPHA by rows or by columns in the specified direction. Rather than having four global labels (very taxing on the ROM FAT), the program uses the rightmost character in ALPHA as the control character to determine the scrolling direction, as follows:

Char$	Direction	Description
"**^**"	Up	moves the rows upwards, first row wraps to last
"**D**"	Down	moves rows downwards, last row wraps to first
"**<**"	Left	moves rows leftwards, first column wraps to last
"**>**"	Right	Right: moves columns rightwards, last column wraps to first

The control char is separated from the matric name by a comma character, for instance: "MNAME,<" will scroll MNAME left. Note that these operations are not specific to complex matrices, and that any other character different from those four will give the "NONEXISTENT" data error.

Program Listing:

```
01  LBL "MSCRL$"          ; "MNAME,chr"
02  E                     ; real matrix
03  XROM "?1X1            ; test case
04  CF 00
05  DIM?                  ; m.00n
06  ASWAP
07  ATOX
08  RDN
09  GTO IND T
10  *LBL 94               "^", upwards
11  INT                   ; m
12  2
13  –                     ; m-2
14  I<>J                  ; 0.00(m-2)
15  GTO 01
16  *LBL 68               ; "D", downwards
17  INT                   ; m
18  RCL X
19  E
20  –                     ; m-1
21  ENTER^
22  I<>J                  ; 0.00(m-1)
23  GTO 02
24  *LBL 62               ; ">", rightwards
25  SF 00
26  FRC                   ; 0.00n
27  2
28  I<>J                  ; 0.002
29  –                     ; 0.00(n-2)
30  *LBL 01
31  2                     ; prepare rows:
32  E3/E+                 ; 1.002
33  *LBL 03
34  XEQ 00
35  +
36  ISG Y
37  GTO 03
38  MNAME?
39  RTN
40  *LBL 60               "<", leftwards
41  SF 00
42  FRC
43  ENTER^
44  I<>J
45  E
46  –
47  RCL X
48  *LBL 02
49  RCL Z
50  +
51  *LBL 04
52  XEQ 00
53  –
54  DSE Y
55  GTO 04
56  MNAME?
57  RTN
58  *LBL 00
59  FC? 00
60  R<>R
61  FS? 00
62  C<>C
63  E
64  E3/E+
65  END
```

You can use the alternative listings below If you prefer individual routine for each action.

```
01 LBL "M^"      ; upwards
02 DIM?          ; n.00m
03 INT           ; n
04 2
05 –             ; n-2
06 I<>J          ; 0.00(n-2) - counter
07 1.002         ; initial pair
08 *LBL 00
09 R<>R          ; swap rows
10 1.001         ; offset
11 +             ; update pointers
12 ISG Y         ; next row
13 GTO 00        ; loop back
14 RTN           ; done.
15 LBL "M_"      ; downwards
16 DIM?          ; n.00m
17 INT
18 RCL X
19 1
20 –             ; n-1 - counter
21 ENTER^
22 I<>J          ; 0.00(n-1)
23 RCL Z         ; n
24 +             ; n.00(n-1)
25 *LBL 02
26 R<>R          ; swap rows
27 1.001         ; offset
28 –             ; update pointers
29 DSE Y         ; next row
30 GTO 02        ; loop back
31 RTN           ; done.
```

```
32 LBL "M>"      ; rightwards
33 DIM?          ; n.00m
34 FRC
35 ,002
36 –             ; 0.00(m-2) - counter
37 1.002         ; initial pair
38 *LBL 01
39 C<>C          ; swap columns
40 1,001         ; offset
41 +             ; update pointers
42 ISG Y         ; next column
43 GTO 01        ; loop back
44 RTN           ; done.
45 LBL "M<"      ; letfwards
46 DIM?          ; n.00m
47 FRC           ; 0.00m
48 ENTER^
49 I<>J          ; m
50 1
51 –             ; m-1 - counter
52 RCL X
53 RCL Z         ; 0.00m
54 +             ; (m-1).00m
55 *LBL 03
56 C<>C          ; swap columns
57 1,001         ; offset
58 –             ; update pointers
59 DSE Y         ; next column
60 GTO 03        ; loop back
61 END           ; done.
```

Note that like in the **MSCRL$** case, the we could combine the termination of routines "**M_**" and "**M<**" (LBL 02 and LBL 03) into a common one using a control flag to determine the row/column case, and ditto with routines "**M^**" and "**M>**" (LBL 00 and LBL 01). That would shorten the program listing and possibly also the byte count, at the expense of using a flag resource.

Appendix: Splicing and Splitting the data blocks.

The **SPLICE** and **SPLIT** routines expect the data block size in the X-register. For instancefor a m x n complex matrix the real and imaginary parts have a size of m.n each, thus the data block size is m.n. In terms of data registers needed, it is twice the size of both blocks together, i.e. 2x (2.m.n) = 4.m.n

See the example below with n=5 complex coefficients, with the real parts in white cells and the imaginary parts in yellow cells. The arrangement on the left is split, and on the right is spliced. The objective is to write routines to move the data between those arrangements.

	SPLICE			SPLIT	
	FOR k = 1 TO n			FOR k = 1 TO n	
	k --> 2n+2k-1			2k-1 --> 2n+k	
	n+k --> 2n+2k			2k --> 3n+k	
	NEXT k			NEXT k	
1	1	R1	1	1	
	2	R2		A	
	3	R3		2	
	4	R4		B	
n	5	R5	n	3	
n+1	A	R6	n+1	C	
	B	R7		4	
	C	R8		D	
	D	R9		5	
2n	E	R10	2n	E	
2n+1	1	R11	2n+1	1	
	A	R12		2	
	2	R13		3	
	B	R14		4	
3n	3	R15	3n	5	
3n+1	C	R16	3n+1	A	
	4	R17		B	
	D	R18		C	
	5	R19		D	
4n	E	R20	4n	E	
	(2n+1),001|(2n) : REGMOVE			(2n+1),001|(2n) : REGMOVE	

The routines follow the pseudo-BASIC loop shown in the picture above, using REGMOVE for the single register copy operation inside of the loop, repeated n-times. The final step is calling REGMOVE again to do a block copy, replacing the original block with the spliced block. Granted this is not the most sophisticated RPN algorithm but it does the job – at the cost of double number of data registers consumption of course.

Program listing:

No matrix functions here, purely auxiliary routines to aid the driver programs:

```
01 LBL "SPLICE"
02 STO 00          ; n
03 E3/E+           ; 1,00n
04 *LBL 00         ;loop #1
05 ENTER^
06 ENTER^
07 ENTER^
08 INT             ; k
09 RCL 00          ; n
10 +               ; n+k
11 ST+ X           ; 2n+2k
12 E
13 -               ; 2n+2k-1
14 I<>J            ; 0,00(2n+2k-1)
15 X<>Y            ; k,00n
16 INT             ; k
17 +               ; k,00(2n+2k-1)
18 REGMOVE         ; real part
19 RCL 00          ; n
20 +               ;(n+k),00(2n+2k-1)
21 E-3             ; 0,001
22 +               ; (n+k),00(2n+2k)
23 REGMOVE         ; Imaginary
24 RDN             ; k,00n
25 ISG X           ; NEXT k
26 GTO 00          ; loop for next
27 GTO 01          ; all done
28 LBL"SPLIT"
29 STO 00
30 E3/E+           ; 1,00n
31 *LBL 02         ; loop #2
32 ENTER^
33 ENTER^
34 ENTER^
35 RCL 00          ; n
36 ST+ X           ,2n
37 X<>Y            ; k,00n
38 INT             ; k
39 +               ; 2n+k
40 I<>J
41 X<>Y            ; k,00n
42 INT             ; k
43 ST+ X           ; 2k
44 E
45 -               ; 2k-1
46 +               ; 2k-1,00( 2n+k)
47 REGMOVE         ; real part
48 RCL 00          ; n
49 E3/E+           ; 1,00n
50 +               ; 2k,00(3n+k)
51 REGMOVE         ; imaginary
52 RDN             ; k,00n
53 ISG X           ; NEXT k
54 GTO 02          ; loop for next
55 *LBL 01         ; common end
56 E
57 E3/E+           ; 1,001
58 RCL 00          ; n
59 ST+ X           ; 2n
60 +               ;2n+1,001
61 LASTX
62 E6
63 /               ; 0,00|(2n)
64 +               ; (2n+1),001|(2n)
65 REGMOVE         ; block move
66 END
```

Lastly let's also list the core routines that do the actual rotation job, **MROT** and **ZMROT** – courtesy of Jean-Marc Baillard. Note how these indeed implement an in-place algorithm for the operation, therefore have a much more efficient usage of memory than the **M<)R** and **ZM<)R** counterparts.

Real Matrix Rotation: Coefficients starting at R09, using R01-R08 for scratch

T = bbb = address of the 1st coefficient > 8
Z = m = Nb of rows > 1
Y = n = Nb of columns > 1
X = k = Nb of rotations > 0

For example, using the matrix below, starting in R09 and if k=2:

1	5	9	13	17
2	6	10	14	18
3	7	11	15	19
4	8	12	16	20

9, ENTER^
4, ENTER^
5, ENTER^
2, XEQ "MROT"

⇒

9	13	17	18	19
5	14	15	11	20
1	10	6	7	16
2	3	4	8	12

```
 1  *LBL "MROT"
 2  STO 07
 3  RDN
 4  STO 02
 5  RDN
 6  STO 01
 7  X<>Y
 8  STO 08
 9  *LBL 12
10  RCL 02
11   E
12   -
13  STO 05
14  RCL 01
15  LASTX
16   -
17  STO 04
18  RCL 08
19  STO 00
20  STO 03
21  *LBL 10
22  RCL 04
23  STO 06
24  RCL IND 03
25  *LBL 01
26  ISG 03
27  CLX
28  X<> IND 03
29  DSE 06
30  GTO 01
31  RCL 05
32  STO 06
33  X<>Y
34  *LBL 02
35  RCL 01
36  ST+ 03
37  X<>Y
38  X<> IND 03
39  DSE 06
40  GTO 02
41  RCL 04
42  STO 06
43  X<>Y
44  *LBL 03
45  DSE 03
46  X<> IND 03
47  DSE 06
48  GTO 03
49  RCL 05
50  STO 06
51  X<>Y
52  *LBL 04
53  DSE 06
54  NOP
55  GTO 00
56  RCL 01
57  ST- 03
58  X<>Y
59  X<> IND 03
60  GTO 04
61  *LBL 00
62  STO IND 00
63  LASTX
64  ST+ 03
65  RCL 03
66  STO 00
67  2
68  ST- 04
69  ST- 05
70  RCL 04
71  X<=0?
72  GTO 00
73  RCL 05
74  X>0?
75  GTO 10
76  *LBL 00
77  DSE 07
78  GTO 12
79  END
```

Complex Matrix Rotation: Coefficients starting at R01, using the status registers for scratch.

Z = m = Nb of rows > 1
Y = n = Nb of columns > 1
X = k = Nb of rotations > 0

172 bytes / SIZE 2.m.n+1

For example, with the matrix below stored in R01-R40 as follows:

R01-R02	R09-R10	R17-R18	R25-R26	R33-R34
R03-R04	R11-R12	R19-R20	R27-R28	R35-R36
R05-R06	R13-R14	R21-R22	R29-R30	R37-R38
R07-R08	R15-R16	R23-R24	R31-R32	R39-R40

```
1+2i9+10i17+18i    25+26i   33+34i
3+4i 11+12i   19+20i    27+28i   35+36i
5+6i 13+14i   21+22i    29+30i   37+38i
7+8i 15+16i    23+24i    31+32i    39+40i
```

4, ENTER^, 5, ENTER^, 2, XEQ "ZMROT" returns in 20 seconds:

```
17+18i25+26i   33+34i   35+36i37+38i
 9+10i   27+28i    29+30i    21+22i 39+40i
 1+2i  19+20i   11+12i    13+14i31+32i
 3+4i    5+6i     7+8i      15+16i 23+24i
```

```
 1  *LBL "ZMROT"
 2  STO O
 3  RDN
 4  STO N
 5  X<>Y
 6  STO M
 7  *LBL 12
 8  RCL N
 9   E
10  STO 00
11   -
12  STO Q
13  RCL M
14  LASTX
15   -
16  STO P
17  *LBL 10
18  RCL P
19  SIGN
20  RCL 00
21  RCL IND X
22  ISG Y
23  CLX
24  RCL IND Y
25  *LBL 01
26  ISG Z
27  CLX
28  X<>Y
29  X<> IND Z
30  ISG Z
31  CLX
32  X<>Y
33  X<> IND Z
34  DSE L
35  GTO 01
36  RCL Q
37  SIGN
38  RDN
39  *LBL 02
40  RCL M
41  ST+ X
42  ST+ T
43  RDN
44  DSE Z
45  X<>Y
46  X<> IND Z
47  ISG Z
48  CLX
49  X<>Y
50  X<> IND Z
51  DSE L
52  GTO 02
53  RCL P
54  SIGN
55  RDN
56  *LBL 03
57  DSE Z
58  DSE Z
59  DSE Z
60  X<>Y
61  X<> IND Z
62  ISG Z
63  CLX
64  X<>Y
65  X<> IND Z
66  DSE L
67  GTO 03
68  RCL Q
69  SIGN
70  RDN
71  *LBL 04
72   RCL M
73  ST+ X
74  ST- T
75  RDN
76  DSE Z
77  X<>Y
78  X<> IND Z
79  ISG Z
80  CLX
81  X<>Y
82  X<> IND Z
83  DSE L
84  GTO 04
85  CLX
86  SIGN
87  ST+ X
88  ST- P
89  ST- Q
90  RCL M
91  ST+ X
92   +
93  ST+ 00
94  RCL P
95   RCL Q
96  X>Y?
97  X<>Y
98  X>0?
99  GTO 10
100 DSE O
101 GTO 12
102 CLA
103 END
```

Appendix. In-place Data splicing. (by Werner)

See: *https://www.hpmuseum.org/forum/thread-14194.html*

Here's a clever approach to the data block splicing problem using an in-place algorithm instead of the sequential method described in the previous pages. The routine only uses the stack, no data registers. It is much shorter and doesn't really take much longer to perform, so a double-down bonus!

Input: data block size in X

```
1.   LBL "SPLC"
2.   E
3.   X<>Y
4.   X=Y?
5.   RTN
6.   ,1
7.   %
8.   2
9.   +
10.  *LBL 02
11.  RCL Y
12.  RCL Y
13.  *LBL 01
14.  ISG T
15.  X<>Y
16.  X<> IND T
17.  X<> IND Z        IND T <>
18.  X<> IND T         IND Z
19.  ISG Z
20.  GTO 01
21.  ISG X
22.  GTO 02
23.  END
```

Three exchanges per data point make it a beauty to behold in action… FOCAL at its best!

END. of this Manual