SANDMATRIX_4
Matrix Extensions for the HP-41

User’s Manual and Quick Reference Guide

Written and programmed by Ángel M. Martin
August 2013
Acknowledgments.-

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

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

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

Published under the GNU software licence agreement.
Table of Contents. - Revision K.

1. Introduction.

   SandMatrix_4 Revision k 5
   Logical next chapter after the SandMath 5
   The many names of Dr. Who 6
   What isn’t included? 12
   Function index at a glance. 15

2. Lower Page Functions in Detail

2.1. SandMatrix 4 Group

   Alpha String Manipulation. 10
   Other functions in header section 11
   The MATRX program 11
   Matrix Polynomial 15
   N-dimensional Vector operations 17
   3D-Vectors mini calculator 18

2.2. Matrix-101

   Setting up a matrix 20
   How a matrix is stored, Matrix Editors. 21
   How to Specify a matrix 22
   Storing and Recalling Matrix Elements 24
   Updated Matrix Editor 25

2.3. Matrix Functions

   Matrix Arithmetic 26

   Major Matrix Operations 27
      LU Decomposition 28
      Working with Complex Matrices 29
      Using Functions with Complex Matrices 31

   Other Matrix Functions (“Utilities”) 34
      Moving and Exchanging sections 34
      Maxima and minima 35
      Norms and Sums 35

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

(c) Ángel Martin - August 2013
3. Upper Page Functions in Detail

3.1. Advanced Matrix

<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>The Enhanced Matrix Editor(s)</td>
<td>37</td>
</tr>
<tr>
<td>New Matrix Utilities / housekeeping</td>
<td>40</td>
</tr>
<tr>
<td>Finding elements / Driver for M*M</td>
<td>43</td>
</tr>
<tr>
<td>Exponential of a Matrix</td>
<td>44</td>
</tr>
<tr>
<td>Logarithm of a Matrix</td>
<td>45</td>
</tr>
<tr>
<td>Square Root of a Matrix</td>
<td>47</td>
</tr>
<tr>
<td>Matrix Integer Powers and Roots</td>
<td>49</td>
</tr>
<tr>
<td>Lie Product</td>
<td>52</td>
</tr>
<tr>
<td>Matrix Trace</td>
<td>53</td>
</tr>
<tr>
<td>Unitary Diagonal</td>
<td>54</td>
</tr>
<tr>
<td>Sum of Diagonal/Crossed element products</td>
<td>55</td>
</tr>
<tr>
<td>Appendix: Square root of a 2x2 matrix</td>
<td>56</td>
</tr>
</tbody>
</table>

3.2. Polynomials and Linear Algebra.

<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eigenvectors and Eigenvalues</td>
<td>57</td>
</tr>
<tr>
<td>Characteristic Polynomial</td>
<td>58</td>
</tr>
<tr>
<td>SOLVE-based implementation</td>
<td>59</td>
</tr>
<tr>
<td>Formula-based 3-Dimensional Case</td>
<td>61</td>
</tr>
<tr>
<td>General n-dimensional case: Faddevv-Leverrier</td>
<td>63</td>
</tr>
<tr>
<td>Jacobi method for Symmetric Matrices</td>
<td>65</td>
</tr>
</tbody>
</table>

3.3. Managing Polynomials

<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Defining and Storing Polynomials</td>
<td>67</td>
</tr>
<tr>
<td>Polynomial Arithmetic</td>
<td>69</td>
</tr>
<tr>
<td>Evaluating and Copying Polynomials</td>
<td>71</td>
</tr>
<tr>
<td>Polynomial Root Finders</td>
<td>72</td>
</tr>
<tr>
<td>Quartic Equation</td>
<td>72</td>
</tr>
<tr>
<td>General case: Bairstow Method</td>
<td>74</td>
</tr>
</tbody>
</table>

3.4. Applications of Polynomials

<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Equations for Curve Fitting programs</td>
<td>77</td>
</tr>
<tr>
<td>Polynomial Interpolation</td>
<td>78</td>
</tr>
<tr>
<td>Prime Factors Decomposition, Totient function</td>
<td>80</td>
</tr>
<tr>
<td>Fitting data to Polynomials</td>
<td>82</td>
</tr>
<tr>
<td>Orthogonal Polynomial Fit</td>
<td>85</td>
</tr>
<tr>
<td>From Poles to Zeros and back</td>
<td>86</td>
</tr>
<tr>
<td>Partial Fractions Expansion</td>
<td>87</td>
</tr>
</tbody>
</table>

Appendix “M” and END. 90
SandMatrix_4 Module - Revision K
Matrix Extensions for the HP-41 System.

1. Introduction.

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

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

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

1.1. The logical next chapter after the SandMath

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

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

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

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

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

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

<table>
<thead>
<tr>
<th>Size</th>
<th>Main</th>
<th>Dependency</th>
<th>Requires</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>8k + 8k</td>
<td>ALGEBRA</td>
<td>Advantage</td>
<td>n/a</td>
<td></td>
</tr>
<tr>
<td>4k + 8k</td>
<td>MATRIX_4k</td>
<td>Advantage</td>
<td>n/a</td>
<td></td>
</tr>
<tr>
<td>4k</td>
<td>POLYN_4k</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4k + 8k</td>
<td>MATRIX_4L4</td>
<td>Advantage</td>
<td>Lib#4</td>
<td></td>
</tr>
<tr>
<td>8k + 4k</td>
<td>Adv_Matrix</td>
<td>POLYN_4k</td>
<td>n/a</td>
<td>Includes SOLVE/INTEG</td>
</tr>
<tr>
<td>8k + 4k</td>
<td>Adv_Matrix4_I</td>
<td>POLYN_4L4</td>
<td>Lib#4</td>
<td>Includes SOLVE/INTEG</td>
</tr>
<tr>
<td>9k</td>
<td>Adv_Matrix4_II</td>
<td></td>
<td>Lib#4</td>
<td>Includes CURVE FIT (*) for EIGEN only</td>
</tr>
<tr>
<td>8k</td>
<td>SandMatrix</td>
<td>SandMath</td>
<td>Lib#4</td>
<td></td>
</tr>
</tbody>
</table>

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

What isn't included?’

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

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

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

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

<table>
<thead>
<tr>
<th>#</th>
<th>Function</th>
<th>Description</th>
<th>Input</th>
<th>Output</th>
<th>Author</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-SNDMTRX 4</td>
<td>Section Header</td>
<td>none</td>
<td>Displays &quot;Order=?&quot;</td>
<td>Ángel Martin</td>
</tr>
<tr>
<td>2</td>
<td>ABSP</td>
<td>Alpha Back Space</td>
<td>Text in Alpha</td>
<td>Last char deleted</td>
<td>W&amp;W GmbH</td>
</tr>
<tr>
<td>3</td>
<td>AIP</td>
<td>Appends integer part</td>
<td>x in X</td>
<td>INT(x) appended to Alpha</td>
<td>Ángel Martin</td>
</tr>
<tr>
<td>4</td>
<td>ASWAP</td>
<td>Alpha Swap</td>
<td>A,B in Alpha</td>
<td>B,A in Alpha</td>
<td>Ángel Martin</td>
</tr>
<tr>
<td>5</td>
<td>CLAC</td>
<td>CLA from Comma</td>
<td>Text in Alpha</td>
<td>Removed from left to comma</td>
<td>W&amp;W GmbH</td>
</tr>
<tr>
<td>6</td>
<td>DOTN</td>
<td>N-dimensional Dot product</td>
<td>cntl words in Y,X</td>
<td>cntl word result in X</td>
<td>JM Baillard</td>
</tr>
<tr>
<td>7</td>
<td>EQT</td>
<td>Displays Curve Equation</td>
<td>Eqn in R00</td>
<td>Writes equation in Alpha</td>
<td>Ángel Martin</td>
</tr>
<tr>
<td>8</td>
<td>SQR?</td>
<td>Tests for Square matrices</td>
<td>Mname in Alpha</td>
<td>Yes.No – Do it true</td>
<td>Ángel Martin</td>
</tr>
<tr>
<td>9</td>
<td>&quot;MATRIX&quot;</td>
<td>&quot;Easy Matrix&quot; Program</td>
<td>Driver for Major Matrix Ops.</td>
<td>Under prgm control</td>
<td>HP Co.</td>
</tr>
<tr>
<td>10</td>
<td>MPOL</td>
<td>Matrix polynomial</td>
<td>Mname in Alpha, Cntl word in X</td>
<td>Calculates P((A))</td>
<td>Ángel Martin</td>
</tr>
<tr>
<td>11</td>
<td>ST&lt;&gt;A</td>
<td>Swaps Alpha/Stack</td>
<td>V1 in Stack, V2 in Alpha</td>
<td>V2 in Stack, V1 in Alpha</td>
<td>Ángel Martin</td>
</tr>
<tr>
<td>12</td>
<td>V*V</td>
<td>3-dimensional Dot product</td>
<td>prompts for coeffs</td>
<td>result in Matrix</td>
<td>Ángel Martin</td>
</tr>
<tr>
<td>13</td>
<td>&quot;3DV&quot;</td>
<td>3D Vectors</td>
<td>Prompts &quot;</td>
<td>V</td>
<td>V* VX&quot;</td>
</tr>
<tr>
<td>14</td>
<td>-CCD MTRX</td>
<td>Section Header</td>
<td>none</td>
<td>Displays &quot;Running...&quot;</td>
<td>Ángel Martin</td>
</tr>
<tr>
<td>15</td>
<td>C&lt;&gt;C</td>
<td>Column exchange (k&lt;&gt;l)</td>
<td>kkk,lll in X</td>
<td>Columns swapped</td>
<td>W&amp;W GmbH</td>
</tr>
<tr>
<td>16</td>
<td>CMAX</td>
<td>Column Maximum</td>
<td>Col# in X, &quot;OP1&quot; in Alpha</td>
<td>Element value in X</td>
<td>W&amp;W GmbH</td>
</tr>
<tr>
<td>17</td>
<td>CNRM</td>
<td>Column Norm</td>
<td>Col# in X, &quot;OP1&quot; in Alpha</td>
<td>column norm in X</td>
<td>W&amp;W GmbH</td>
</tr>
<tr>
<td>18</td>
<td>CSUM</td>
<td>Column Sum</td>
<td>&quot;OP1,RES&quot; in Alpha</td>
<td>Sum of Cols in RES matrix</td>
<td>W&amp;W GmbH</td>
</tr>
<tr>
<td>19</td>
<td>DIM?</td>
<td>Matrix Dimension</td>
<td>&quot;OP1&quot; in Alpha</td>
<td>dimension placed in X</td>
<td>W&amp;W GmbH</td>
</tr>
<tr>
<td>20</td>
<td>FNRM</td>
<td>Frobenius Norm</td>
<td>&quot;OP1&quot; in Alpha</td>
<td>value in X</td>
<td>W&amp;W GmbH</td>
</tr>
<tr>
<td>21</td>
<td>I+</td>
<td>Increase row index</td>
<td>&quot;OP1&quot; in Alpha</td>
<td>increased i</td>
<td>HP Co.</td>
</tr>
<tr>
<td>22</td>
<td>I-</td>
<td>Decrease row index</td>
<td>&quot;OP1&quot; in Alpha</td>
<td>decreased i</td>
<td>HP Co.</td>
</tr>
<tr>
<td>23</td>
<td>J+</td>
<td>Increase column index</td>
<td>&quot;OP1&quot; in Alpha</td>
<td>increased j</td>
<td>HP Co.</td>
</tr>
<tr>
<td>24</td>
<td>J-</td>
<td>Decrease column index</td>
<td>&quot;OP1&quot; in Alpha</td>
<td>decreased j</td>
<td>HP Co.</td>
</tr>
<tr>
<td>25</td>
<td>M*M</td>
<td>Matrix Product</td>
<td>&quot;OP1,OP2, RES&quot; in Alpha</td>
<td>matrix product in RES</td>
<td>W&amp;W GmbH</td>
</tr>
<tr>
<td>26</td>
<td>MAT*</td>
<td>element multiplication</td>
<td>value in X, &quot;OP1,X&quot; in Alpha</td>
<td>aij = aij * x</td>
<td>W&amp;W GmbH</td>
</tr>
<tr>
<td>27</td>
<td>MAT+</td>
<td>addition of scalar</td>
<td>value in X, &quot;OP1,X&quot; in Alpha</td>
<td>aij = aij + x</td>
<td>W&amp;W GmbH</td>
</tr>
<tr>
<td>28</td>
<td>MAT-</td>
<td>element subtraction</td>
<td>value in X, &quot;OP1,X&quot; in Alpha</td>
<td>aij = aij - x</td>
<td>W&amp;W GmbH</td>
</tr>
<tr>
<td>29</td>
<td>MAT/</td>
<td>Division by scalar</td>
<td>value in X, &quot;OP1,X&quot; in Alpha</td>
<td>aij / aij</td>
<td>W&amp;W GmbH</td>
</tr>
<tr>
<td>30</td>
<td>MATDIM</td>
<td>Dimensions a matrix</td>
<td>mmm,n,m in X, &quot;OP1&quot; in Alpha</td>
<td>Matrix Dimensioned</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>MAX</td>
<td>Maximum element</td>
<td>&quot;OP1&quot; in Alpha</td>
<td>Element value in X</td>
<td>W&amp;W GmbH</td>
</tr>
<tr>
<td>32</td>
<td>MAXAB</td>
<td>Absolute maximum</td>
<td>&quot;OP1&quot; in Alpha</td>
<td>Element value in X</td>
<td>W&amp;W GmbH</td>
</tr>
<tr>
<td>33</td>
<td>MDET</td>
<td>Determinant</td>
<td>&quot;OP1&quot; in Alpha</td>
<td>Determinant in X</td>
<td>HP Co.</td>
</tr>
<tr>
<td>34</td>
<td>MIN</td>
<td>Minimum element</td>
<td>&quot;OP1&quot; in Alpha</td>
<td>minimum element in X</td>
<td>W&amp;W GmbH</td>
</tr>
<tr>
<td>35</td>
<td>MINV</td>
<td>Inverse Matrix</td>
<td>&quot;OP1&quot; in Alpha</td>
<td>Matrix replaced w/ Inverse</td>
<td>HP Co.</td>
</tr>
<tr>
<td>36</td>
<td>MMOVE</td>
<td>Moves part of a matrix</td>
<td>li; kj; b,m,n in XYZ</td>
<td>Elements moved</td>
<td>W&amp;W GmbH</td>
</tr>
<tr>
<td>37</td>
<td>MNAME</td>
<td>Get current Mname to Alpha</td>
<td>none</td>
<td>Matrix Name in Alpha</td>
<td>W&amp;W GmbH</td>
</tr>
<tr>
<td>38</td>
<td>MR</td>
<td>Recall element from pt</td>
<td>none</td>
<td>value in X</td>
<td>HP Co.</td>
</tr>
<tr>
<td>39</td>
<td>MRC+</td>
<td>Recall and advance in Column</td>
<td>&quot;OP1&quot; in Alpha</td>
<td>element in X, increased i</td>
<td>W&amp;W GmbH</td>
</tr>
<tr>
<td>40</td>
<td>MRC-</td>
<td>Recall and back one in Column</td>
<td>&quot;OP1&quot; in Alpha</td>
<td>element in X, decreased i</td>
<td>W&amp;W GmbH</td>
</tr>
<tr>
<td>41</td>
<td>MRIJ</td>
<td>Recall ij pointer of current</td>
<td>none</td>
<td>pointer in X</td>
<td>W&amp;W GmbH</td>
</tr>
<tr>
<td>42</td>
<td>MRIJA</td>
<td>Recall ij pointer of Alpha</td>
<td>&quot;OP1&quot; in Alpha</td>
<td>pointer in X</td>
<td>W&amp;W GmbH</td>
</tr>
<tr>
<td>43</td>
<td>MRR+</td>
<td>Recall and advance in Row</td>
<td>&quot;OP1&quot; in Alpha</td>
<td>element in X, increased j</td>
<td>W&amp;W GmbH</td>
</tr>
<tr>
<td>44</td>
<td>MRR-</td>
<td>Recall and back one in Row</td>
<td>&quot;OP1&quot; in Alpha</td>
<td>element in X, decreased j</td>
<td>W&amp;W GmbH</td>
</tr>
<tr>
<td>45</td>
<td>MS</td>
<td>Store element at pointer</td>
<td>value in X, &quot;OP1&quot; in Alpha</td>
<td>Element stored</td>
<td>HP Co.</td>
</tr>
<tr>
<td>46</td>
<td>MSC+</td>
<td>Store and advance in Column</td>
<td>value in X, &quot;OP1&quot; in Alpha</td>
<td>element stored, increased i</td>
<td>W&amp;W GmbH</td>
</tr>
<tr>
<td>47</td>
<td>MSJ</td>
<td>Sets pointer of current matrix</td>
<td>ijj in X</td>
<td>pointer set</td>
<td>W&amp;W GmbH</td>
</tr>
<tr>
<td>48</td>
<td>MSJA</td>
<td>Sets points of Matrix in Alpha</td>
<td>ijj in X, &quot;OP1&quot; in Alpha</td>
<td>pointer set</td>
<td>W&amp;W GmbH</td>
</tr>
<tr>
<td>49</td>
<td>MSR+</td>
<td>Store and advance in Row</td>
<td>value in X, &quot;OP1&quot; in Alpha</td>
<td>element stored, increased j</td>
<td>W&amp;W GmbH</td>
</tr>
<tr>
<td>50</td>
<td>MSWAP</td>
<td>Swaps part of a matrix</td>
<td>li; kj; b,m,n in XYZ</td>
<td>Elements Swapped</td>
<td>W&amp;W GmbH</td>
</tr>
<tr>
<td>51</td>
<td>MSYS</td>
<td>Linear Systems</td>
<td>&quot;OP1,OP2, RES&quot; in Alpha</td>
<td>Resolves Linear System</td>
<td>HP Co.</td>
</tr>
<tr>
<td>52</td>
<td>PIV</td>
<td>Sets pointer to pivot element</td>
<td>Col# in X, &quot;OP1&quot; in Alpha</td>
<td>Element value in X</td>
<td>W&amp;W GmbH</td>
</tr>
<tr>
<td>#</td>
<td>Function</td>
<td>Description</td>
<td>Input</td>
<td>Output</td>
<td>Author</td>
</tr>
<tr>
<td>----</td>
<td>----------</td>
<td>-------------</td>
<td>-------</td>
<td>--------</td>
<td>--------</td>
</tr>
<tr>
<td>53</td>
<td>R&lt;&gt;R</td>
<td>Row Exchange (k&lt;&gt;l)</td>
<td>kkk,ll in X</td>
<td>Rows swapped</td>
<td>W&amp;W GmbH</td>
</tr>
<tr>
<td>54</td>
<td>R&lt;&gt;?</td>
<td>Row comparison test</td>
<td>kkk,ll in X</td>
<td>skip line if false</td>
<td>W&amp;W GmbH</td>
</tr>
<tr>
<td>55</td>
<td>RMAXAB</td>
<td>Absolute maximum</td>
<td>row# in X, OP1 in Alpha</td>
<td>element in X, pointer to ij</td>
<td>W&amp;W GmbH</td>
</tr>
<tr>
<td>56</td>
<td>RNRM</td>
<td>Row Norm</td>
<td>&quot;OP1&quot; in Alpha</td>
<td>Row Norm in X</td>
<td>W&amp;W GmbH</td>
</tr>
<tr>
<td>57</td>
<td>RSUM</td>
<td>Row Sum</td>
<td>&quot;OP1,RES&quot; in Alpha</td>
<td>sums of rows in RES matrix</td>
<td>W&amp;W GmbH</td>
</tr>
<tr>
<td>58</td>
<td>SUM</td>
<td>Element Sum</td>
<td>&quot;OP1&quot; in Alpha</td>
<td>element sum in X</td>
<td>W&amp;W GmbH</td>
</tr>
<tr>
<td>59</td>
<td>SUMAB</td>
<td>Absolute Values Sum</td>
<td>&quot;OP1&quot; in Alpha</td>
<td>element absolute sum in X</td>
<td>W&amp;W GmbH</td>
</tr>
<tr>
<td>60</td>
<td>TRNPS</td>
<td>Transpose</td>
<td>&quot;OP1&quot; in Alpha</td>
<td>Matrix replaced w/ transposed</td>
<td>HP Co.</td>
</tr>
<tr>
<td>61</td>
<td>YC+C</td>
<td>Adds Y*Col (l) to Col (k)</td>
<td>value in y, kkk,ll in X</td>
<td>column k changed</td>
<td>W&amp;W GmbH</td>
</tr>
<tr>
<td>62</td>
<td>&quot;MEDIT&quot;</td>
<td>Matrix Editor</td>
<td>prompts for elements</td>
<td>Edits Matrix</td>
<td>HP Co.</td>
</tr>
<tr>
<td>63</td>
<td>&quot;CMEDIT&quot;</td>
<td>Complex Matrix Editor</td>
<td>prompts for coeffs</td>
<td>Edits Complex matrix</td>
<td>HP Co.</td>
</tr>
<tr>
<td>64</td>
<td>MPT</td>
<td>Matrix Prompt</td>
<td>i,j,jj in x</td>
<td>Prompts for element</td>
<td>Ángel Martin</td>
</tr>
</tbody>
</table>

### -ADV MATRIX Section Header
- none
- Displays "Not Square" | Ángel Martin |

### -MROW
- Input Row | "OP1" in Alpha, row# in x | Prompts for Row | Ángel Martin |
### i<>J
- Swaps indexes | i,j,jj in X | j,00i in X, i00j in LastX | Ángel Martin |
### i?j?
- Is i ≠ j? | i,j,jj in X | comparison, skip if False | Ángel Martin |
### IMC
- Input Matrix by Columns | "OP1" in Alpha | Inputs elements by columns | Ángel Martin |
### IMR
- Input Matrix by Rows | "OP1" in Alpha | Inputs elements by rows | Ángel Martin |
### LU?
- Tests for L/U Decomposed | Mname in Alpha | Yes.No – Do it true | Ángel Martin |
### M+1/X
- x-th. root of a Matrix | "OP1" in Alpha, x in X | Matrix replaced by its root | Ángel Martin |
### M+2
- Matrix Square | "OP1" in Alpha | Matrix replaced by [M][M] | Ángel Martin |
### MAT=Copy Matrix | "OP1,RES" in Alpha | Copies matrix A into B | Ángel Martin |
### MATP
- Driver for M*M | Driver for M*M | Under prgm control | Ángel Martin |
### MCON
- Constant | "OP1" in Alpha, x in X | Makes all elements =x | Ángel Martin |
### MDPS
- Diagonal Product Sum | "OP1" in Alpha | Sum of diagonal products | Ángel Martin |
### "MEXP"
- Matrix Exponential | "OP1" in Alpha | Matrix replaced by exp(M) | Ángel Martin |
### MFIND
- Element finder | "OP1" in Alpha, x in X | Element pointer if found | Ángel Martin |
### MIDN
- Identity Matrix | "OP1" in Alpha | Makes it Identity Matrix | Ángel Martin |
### MLIE
- Matrix Lie Product | "OP1,OP2,RES" in Alpha | [A][B] - [B][A] | Ángel Martin |
### MLN
- Matrix Natural Log | "OP1" in Alpha | Matrix replaced by LN(M) | Ángel Martin |
### MPWR
- Matrix Power to X | "OP1" in Alpha, x in X | Matrix replaced by (M)^xN(x) | Ángel Martin |
### MRDIM
- Matrix Redimension | "OP1" in Alpha, dim in X | Matrix redimensioned | Ángel Martin |
### MSQRT
- Matrix Square Root | "OP1" in Alpha | Matrix replaced by SRT([M]) | Ángel Martin |
### MSORT
- Sorts matrix elements | "OP1" in Alpha | Matrix Elements sorted | Ángel Martin |
### MSZE?
- Matrix Size | "OP1" in Alpha | Matrix size in X | Ángel Martin |
### MTRACE
- Matrix Trace | "OP1" in Alpha | Trace in x | Ángel Martin |
### MZERO
- Zeros a Matrix | "OP1" in Alpha | All elements zeroed | Ángel Martin |
### OMC
- Output Matrix by Columns | "OP1" in Alpha | Shows elements by columns | Ángel Martin |
### OMR
- Output Matrix by Rows | "OP1" in Alpha | Shows elements by rows | Ángel Martin |
### OX
- Output x-th column | "OP1" in Alpha, Col# in X | Shows Col elements | Ángel Martin |
### ORX
- Output x-th row | "OP1" in Alpha, Row# in X | Shows Row elements | Ángel Martin |
### PMTM
- Prompts for Matrix | "OP1" in Alpha | Prompts for complete Rows | Ángel Martin |
### R/aRR
- Unitary Diagonal | "OP1" in Alpha | Diagonal elements = 1 | Ángel Martin |
### Sigma
- Sum of crossed products | "OP1" in Alpha | \(\sum_{ij} i|^2j|\) in X | Ángel Martin |

### -ADV POLYN Section Header
- none
- Displays "\(\sum_{ij} x^i y^j\)" | Ángel Martin |

### "BRSTW"
- Bairsrow Method | Cntl word in Z, guesses in Y,X | shows results | JM Baillard |
### CHRPOL
- Characteristic Polynomial | Under prgm control | Characteristic Pol Coefs | Ángel Martin |
### DTC
- Detele Tiny Coefficients | Cntl word in X | Deletes ak < 1E-7 | JM Baillard |
### EIGEN
- Eigen Values by SOLVE | Under prgm control | Eigen Values by Solve | Ángel Martin |
### EV3
- Eigen values 3x3 | Matrix in XMEM | Eigen Values by Formula | Ángel Martin |
### EV3X3
- Eigen values 3x3 | Prompts Matrix Elements | Eigen Values by Formula | Ángel Martin |
### JACOBI
- Symmetrical Eigenvalues | Under prgm control | Eigen Values by Jacobi | Valentín Albillo |
### OPFIT
- Orthogonal polynomial Fit | Under prgm control | shows results | Eugenio Úbeda |
### "P+P"
- Polynomial Addition | Driver for PSUM w/CF 01 | shows results | Ángel Martin |
### "P-P"
- Polynomial Subtraction | Driver for PSUM w/SF 01 | shows results | Ángel Martin |
### "P+P"
- Polynomial Multiplication | Driver for PPRD | shows results | Ángel Martin |

(c) Ángel Martin - August 2013  Page 8
<table>
<thead>
<tr>
<th>#</th>
<th>Function</th>
<th>Description</th>
<th>Input</th>
<th>Output</th>
<th>Author</th>
</tr>
</thead>
<tbody>
<tr>
<td>45</td>
<td>&quot;P/P&quot;</td>
<td>Polynomial Division</td>
<td>Driver for PDIV</td>
<td>shows results</td>
<td>Ángel Martin</td>
</tr>
<tr>
<td>46</td>
<td>PCPY</td>
<td>Copy of Polynomial</td>
<td>from, to cntl words in Y,X</td>
<td>polynomial copied</td>
<td>JM Baillard</td>
</tr>
<tr>
<td>47</td>
<td>PDIV</td>
<td>Euclidean Division</td>
<td>cont words in Y and X</td>
<td>cntl words remainder &amp; result</td>
<td>JM Baillard</td>
</tr>
<tr>
<td>48</td>
<td>PEDIT</td>
<td>Polynomial Editor</td>
<td>cntl word in X</td>
<td>prompts for each coeff value</td>
<td>Ángel Martin</td>
</tr>
<tr>
<td>49</td>
<td>PFE</td>
<td>Partial Fraction Expansion</td>
<td>Under prgm control</td>
<td>see description to decode</td>
<td>JM Baillard</td>
</tr>
<tr>
<td>50</td>
<td>&quot;PF&gt;X&quot;</td>
<td>Prime Factors to Number</td>
<td>Matrix w/ Prime Facts in XMEM</td>
<td>restores the original argument</td>
<td>Ángel Martin</td>
</tr>
<tr>
<td>51</td>
<td>PMP</td>
<td>Prompts for Polynomial</td>
<td>cntl word in X</td>
<td>prompts for complete list</td>
<td>Ángel Martin</td>
</tr>
<tr>
<td>52</td>
<td>POLFIT</td>
<td>Polynomial Fit</td>
<td>Under prgm control</td>
<td>calculates polynomial fit</td>
<td>Valentin Albillo</td>
</tr>
<tr>
<td>53</td>
<td>POLINT</td>
<td>Aitken Interpolation</td>
<td>Under prgm control</td>
<td>interpolation made</td>
<td>Ulrich Deiters</td>
</tr>
<tr>
<td>54</td>
<td>POLZER</td>
<td>From Poles to Zeros</td>
<td>Under prgm control</td>
<td>shows coeffs and roots</td>
<td>Ángel Martin</td>
</tr>
<tr>
<td>55</td>
<td>PPRD</td>
<td>Polynomial Product</td>
<td>cntl words in Z, Y, bbb in X</td>
<td>cntl word result in X</td>
<td>JM Baillard</td>
</tr>
<tr>
<td>56</td>
<td>&quot;PRMF&quot;</td>
<td>Prime Factors Decomposition</td>
<td>number in X</td>
<td>prime factors in XMEM Matrix</td>
<td>Ángel Martin</td>
</tr>
<tr>
<td>57</td>
<td>&quot;PROOT&quot;</td>
<td>Polynomial Roots</td>
<td>Under prgm control</td>
<td>Shows all roots</td>
<td>Ángel Martin</td>
</tr>
<tr>
<td>58</td>
<td>PSUM</td>
<td>Polynomial Sum</td>
<td>cntl words in Y, Z, bbb in X</td>
<td>cntl word result in X</td>
<td>JM Baillard</td>
</tr>
<tr>
<td>59</td>
<td>PVAL</td>
<td>Polynomial Evaluation</td>
<td>Cntl word in Y, x in X</td>
<td>Result in X</td>
<td>JM Baillard</td>
</tr>
<tr>
<td>60</td>
<td>PVVIEW</td>
<td>Polinomial View</td>
<td>Cntl word in X</td>
<td>Sequential listing of coeffs</td>
<td>Ángel Martin</td>
</tr>
<tr>
<td>61</td>
<td>QUART</td>
<td>Quartic Equation Roots</td>
<td>coeef in Stack (a4=1)</td>
<td>shows results</td>
<td>JM Baillard</td>
</tr>
<tr>
<td>62</td>
<td>&quot;RTSN&quot;</td>
<td>Roots subroutine</td>
<td>Under prgm control</td>
<td>calculates roots</td>
<td>Ángel Martin</td>
</tr>
<tr>
<td>63</td>
<td>TOTNT</td>
<td>Euler’s Totient Function</td>
<td>argument in X</td>
<td>Result in X</td>
<td>Ángel Martin</td>
</tr>
<tr>
<td>64</td>
<td>&quot;#EV&quot;</td>
<td>Subroutine for EIGEN</td>
<td>Under prgm control</td>
<td>Under prgm control</td>
<td>Ángel Martin</td>
</tr>
</tbody>
</table>

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

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

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

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

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

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

(c) Ángel Martin - August 2013
2. Lower-Page Functions in detail

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

### 2.1. Alpha String Management

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

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

<table>
<thead>
<tr>
<th>#</th>
<th>Function</th>
<th>Description</th>
<th>Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><strong>ABSP</strong></td>
<td>Alpha Back Space</td>
<td>Text in Alpha</td>
</tr>
<tr>
<td>2</td>
<td><strong>AIP</strong></td>
<td>Appends integer part</td>
<td>x in X</td>
</tr>
<tr>
<td>3</td>
<td><strong>ASWAP</strong></td>
<td>Alpha Swap</td>
<td>A,B in Alpha</td>
</tr>
<tr>
<td>4</td>
<td><strong>CLAC</strong></td>
<td>CLA from Comma</td>
<td>Text in Alpha</td>
</tr>
<tr>
<td>5</td>
<td><strong>EQT</strong></td>
<td>Displays Curve Equation</td>
<td>Eq# in R00 (1 – 16)</td>
</tr>
<tr>
<td>6</td>
<td><strong>ST&lt;&gt;A</strong></td>
<td>Exchanges Alpha and Stack</td>
<td>Values in Stack and Alpha registers</td>
</tr>
</tbody>
</table>

**ABSP** deletes the rightmost character in ALPHA – equivalent to “back space” in manual mode.

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

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

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

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

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

<table>
<thead>
<tr>
<th>#</th>
<th>Function</th>
<th>Description</th>
<th>Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>&quot;MATRX&quot;</td>
<td>&quot;Easy Matrix&quot; Program</td>
<td>Driver for Major Matrix Ops.</td>
</tr>
<tr>
<td>2</td>
<td>SQR?</td>
<td>Tests for Square Matrix</td>
<td>Mname in Alpha</td>
</tr>
<tr>
<td>3</td>
<td>MPOL</td>
<td>Matrix polynomial</td>
<td>Mname in Alpha, Cnt'l word in X</td>
</tr>
<tr>
<td>4</td>
<td>DOTN</td>
<td>N-dimensional Dot product</td>
<td>cnt'l words in Y,X</td>
</tr>
<tr>
<td>5</td>
<td>V*V</td>
<td>3-dimensional Dot product</td>
<td>prompts for coeffs</td>
</tr>
<tr>
<td>6</td>
<td>&quot;3DV&quot;</td>
<td>3D Vectors</td>
<td>Prompts &quot;</td>
</tr>
</tbody>
</table>

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

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

### 2.2.1 The Matrix Program

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

What this program can do.

Consider the equations:

\[
\begin{align*}
3.8 x_1 + 7.2 x_2 &= 16.5 \\
1.3 x_1 - 0.9 x_2 &= -22.1
\end{align*}
\]

for which you must determine the values of \(x_1\) and \(x_2\). These equations can be expressed in matrix form as \(AX = B\), where \(A\) is the coefficient matrix for the system, \(B\) is the column or constant matrix, and \(X\) is the solution or result matrix.

\[
A = \begin{bmatrix} 3.8 & 7.2 \\ 1.3 & -0.9 \end{bmatrix}, \quad X = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}, \quad B = \begin{bmatrix} 16.5 \\ -22.1 \end{bmatrix}
\]

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

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

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

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

![Initial Menu](image1)

![Main Menu](image2)

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

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

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

Remarks

**Alteration of the Original Matrix.** The input matrix A is altered by the operations finding the inverse, the determinant, the transpose and the solution of the matrix equation. You can re-invert $A^{-1}$, and re-transpose $A^T$ to restore the original form of $A$. However, if you have calculated the determinant or the solution matrix, then $A$ is in its LU-decomposed form. To restore $A$, simply **invert it twice**. The LU-decomposition does not interfere with any subsequent **MATRIX** operation except transposition and editing (do not attempt to edit an LU-decomposed matrix unless you intend to change every element). For more information on LU-decomposition, refer to "LU-Decomposition" in the next chapter ("Matrix Functions").
Matrix Storage. The MATRX program stores a matrix $A$ starting in R0 of main memory; it is named "$R0$". Its column matrix $B$ is stored after it, and the result matrix $X$ overwrites $B$, Refer to the chapter 'Matrix Functions' for an explanation of how matrices are named and stored, and how much room they need. MATRX cannot access any other matrices, with the exception of the previous $R0$ and its corresponding column matrix.

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

Example 1.

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

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

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

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

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

Keystrokes | Display | Comments |
---|---|---|
XEQ “MATRIX” | “RL CX” | Starts the MTRX program |
[B] (CX) | “ORDER=?” | Selects a complex Matrix |
2, R/S | “A I DT B SE” | Dimensions a 2x2 complex matrix |
[A], R/S | “RE1:1=x11?” | Enters the editor and displays old value |
1, R/S | “IM1:1=y11?” | ditto for the imaginary part |
2, R/S | “RE1:2=x12?” |
3, R/S | “IM1:2=y12?” |
4, R/S | “RE:2:1=x21?” | Wrong entry! Should be 3, not 4... |
1,002, [A] | “RE1:2=3.000?” | Moves editor back to x12 |
R/S | “IM1:2=4.000?” | The wrong imaginary part |
3, R/S | “RE2:1=x21?” | Correct value is entered for y12. Proceed |
4, R/S | “IM2:1=y21?” |
5, R/S | “RE2:2=x223?” |
6, R/S | “IM2:2=y227?” |
7, R/S | “A I DT B SE” | Enters last element and returns main menu |
[B] (I) | “A I DT B SE” | Inverts A |
[SHIFT][A] | “RE1:1=-0.9663” | Viewing A-1 |
2.002, [A] | “RE2:2=-0.2369” | Displays x22 + iy22 |
R/S | “IM2:2=-0.0225” |
R/S (or [J]) | “A I DT B SE” | Exits the editor |

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

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

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

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

\[
P(x) = 2x^4 - x^3 + 3x^2 - 4x + 5 \; \text{and:} \\
A = \begin{bmatrix} 4 & 2 & 3 \\ 3 & 2 & 5 \\ 2 & 1 & 4 \end{bmatrix}
\]

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

\[
\text{ALPHA, "A", ALPHA, 3,003, XEQ "MATDIM"}
\]

\[
\text{XEQ "PMTM" -> at the prompt "R1: _" we type: 4, ENTER^, 2, ENTER^, 3, R/S} \\
\text{-> at the prompt "R2: _" we type: 3, ENTER^, 2, ENTER^, 5, R/S} \\
\text{-> at the prompt "R3: _" we type: 2, ENTER^, 1, ENTER^, 4, R/S}
\]

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

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

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

\[
2.006, \text{XEQ "PMTP" -> at the prompt "R2: _" we type:} \\
2, \text{ENTER^, CHS, 1, ENTER^, 3, ENTER^, CHS, 4, ENTER^, 5, R/S}
\]

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

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

\[
P(A) = \begin{bmatrix} 3548 & 1887 & 4705 \\ 3727 & 1987 & 4962 \\ 2539 & 1351 & 3385 \end{bmatrix}
\]
The result matrix name is placed in ALPHA when the execution ends, so you can directly use any matrix editor routine (like OMR) to review its elements. Note how OMR will display integer values without any zeros after the decimal point, regardless of the current FIX settings. Set flag 21 to stop the display of each individual element.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

<table>
<thead>
<tr>
<th>STACK</th>
<th>INPUTS</th>
<th>OUTPUTS</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>/</td>
<td></td>
<td>46.52626239°</td>
</tr>
<tr>
<td>Z</td>
<td>/</td>
<td></td>
<td>( | U | ) 7.874007874</td>
</tr>
<tr>
<td>Y</td>
<td>bbb.eee(U)</td>
<td>( | V | ) 7.937253933</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>bbb.eee(V)</td>
<td>U,V 43,000000</td>
<td></td>
</tr>
<tr>
<td>L</td>
<td>/</td>
<td>( \cos \mu )</td>
<td></td>
</tr>
</tbody>
</table>

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

The alternative - Vectors as Matrices.

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

To appreciate the workings of \( V^*V \) you need to consider that it transposes \( V_2 \) before doing the multiplication, and that it calculates the Frobenius norms of each matrix on the fly to obtain the angle. The dot product is placed in a 1x1 matrix named “\( V^*V \)” in X-Mem.

Here's the listing of the program that clearly shows all the housekeeping chores required to prepare the strings needed in ALPHA for the matrix functions as input. Even if it's somehow slower and less efficient, it's a good “academic example” of utilization of the standard matrix functions.
The usage of user flag 06 determines whether the program is used as a subroutine – in which case the data entry is skipped. This is more or less consistently done throughout the SandMatrix module, and has the benefit of saving one entry in the FAT – which would be needed for the subroutine label.

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

### 2.2.4.- 3D Vectors Mini-Calculator.

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

Its prompt looks like this:

![3DV Prompt](image)

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

Data entry is also under program control, and nice alphanumeric mnemonics describe the result(s). The module and the dot product are left in X upon completion. For the cross product case the three components are sequentially displayed, with a pause in between them. They’re also placed in the stack registers Z,Y,X for subroutine use.
The program listing is below – note how this trivial little application manages to make good use of some of the sub-functions in the SandMath module, as well as the interesting way to use the ALPHA register for the vector components.

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

You’re encouraged to check the **Vector Analysis ROM** for a comprehensive implementation of a 3D-Vector calculator, as well as other geometry programs. The Vectors ROM is completely self-contained, and only takes up one page (4k), complementing the SandMatrix (and the SandMath) very effectively.
2.2. MATRIX 101

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

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

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Inputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 MATDIM</td>
<td>Dimensions a Matrix</td>
<td>Name in Alpha, dimensions in X</td>
</tr>
<tr>
<td>2 MNAME?</td>
<td>Returns name of current Matrix to Alpha</td>
<td>none</td>
</tr>
<tr>
<td>3 DIM?</td>
<td>Returns the dimension of Matrix</td>
<td>Name in Alpha</td>
</tr>
<tr>
<td>4 &quot;MEDIT&quot;</td>
<td>Matrix Editor</td>
<td>Name in Alpha</td>
</tr>
<tr>
<td>5 &quot;CMEDIT&quot;</td>
<td>Complex Matrix Editor</td>
<td>Name in Alpha</td>
</tr>
</tbody>
</table>

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

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

Naming a Matrix

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

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

Dimensioning a Matrix

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

- **MNAME?** Returns the name of the current matrix to the Alpha register.

- **DIM?** Returns the dimensions mmm.nnn of the matrix specified in the Alpha register to the X-register. (A blank Alpha register specifies the current matrix.)

**How a Matrix Is Stored**

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

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

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

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

Redimensioning 2 x 3 to 2 x 2:

\[
\begin{bmatrix}
1 & 2 & 3 \\
4 & 5 & 6
\end{bmatrix} \rightarrow \begin{bmatrix}
1 & 2 \\
3 & 4
\end{bmatrix}, \text{ lost}
\]

Redimensioning 2 x 3 to 2 x 4:

\[
\begin{bmatrix}
1 & 2 & 3 \\
4 & 5 & 6
\end{bmatrix} \rightarrow \begin{bmatrix}
1 & 2 & 3 & 4 \\
5 & 6 & \_ & \_
\end{bmatrix}
\]

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

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

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

**Using the Matrix Editors**

There are two matrix editors: **MEDIT** for real matrices and **CMEDIT** for complex matrices. They are otherwise quite similar. The matrix editors are used for three purposes:
• Entering new values into the elements of a matrix.
• Reviewing and changing (editing) the elements of a matrix, either in order or by “random access” of specific elements.
• Viewing (without being able to change) the elements of a matrix (flag 08 set).

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

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

**Viewing without editing.**- If you set flag 08, the editor will let you only view the elements, not change them. The display appears without the “?” at the end of the line. 1:1= 1.0000

If you have a printer attached while flag 08 is set, it will print out all the elements of the matrix without pausing.

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

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

**How to Specify a Matrix**

Given the matrix multiplication operation \( AB = C \), you know \( A \) and \( B \) and are looking for the product matrix, \( C \). In performing this operation, the calculator must be given the identities of the existing matrices \( A \) and \( B \), and also be told where to put the result matrix, \( C \). (However, the result matrix can be the same as one of the input matrices.) All given matrices must already exist as named, dimensioned matrices. Naturally, only \( A \) and \( B \) must contain valid data.

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

A matrix function checks the Alpha register for the names (that is, the locations) of the matrices it needs for input and output. Before executing that function, you should specify all needed parameters on one line in the Alpha register, separating each with a comma:
**Scalar Operations.** - Scalar input and output must be in the X-register, and so this location does not need to be specified unless the function in question can use *either* a scalar *or* a matrix for the same input parameter. To specify the X-register, use X.

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

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

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

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

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

<table>
<thead>
<tr>
<th>Alpha Register’s Contents</th>
<th>Matrices Specified</th>
</tr>
</thead>
<tbody>
<tr>
<td>A,B,C</td>
<td>A, B, C</td>
</tr>
<tr>
<td>A,B</td>
<td>A, B, B</td>
</tr>
<tr>
<td>A</td>
<td>A, A, A</td>
</tr>
<tr>
<td>A,,B</td>
<td>A, A, B</td>
</tr>
<tr>
<td>,A,B</td>
<td>current, A, B</td>
</tr>
<tr>
<td>,A</td>
<td>current, A, A</td>
</tr>
<tr>
<td>,,A</td>
<td>current, current, A</td>
</tr>
<tr>
<td>X,A,B</td>
<td>X-reg, A, B</td>
</tr>
<tr>
<td>X,A</td>
<td>X-reg, A, A</td>
</tr>
<tr>
<td>A,X</td>
<td>A, X-reg, A</td>
</tr>
<tr>
<td>A,,X</td>
<td>A, A, A (ignores X)</td>
</tr>
<tr>
<td>X</td>
<td>X-reg, current, current</td>
</tr>
<tr>
<td>(blank)</td>
<td>current, current, current</td>
</tr>
</tbody>
</table>
2.2.2.- Storing and Recalling individual Matrix elements.

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

**Setting and recalling the Pointer**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Inputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 MSIJA</td>
<td>Sets element pointer of matrix in Alpha</td>
<td>Name in Alpha, ( iii,jjj ) in X-reg.</td>
</tr>
<tr>
<td>2 MSIJ</td>
<td>Sets element pointer of current matrix</td>
<td>( iii,jjj ) in X-reg.</td>
</tr>
<tr>
<td>3 MRIJA</td>
<td>Recalls element pointer of Matrix in Alpha</td>
<td>Name in Alpha, ( iii,jjj ) in X-reg.</td>
</tr>
<tr>
<td>4 MRIJ</td>
<td>Recalls element pointer of current matrix</td>
<td>( iii,jjj ) in X-reg.</td>
</tr>
</tbody>
</table>

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

**Incrementing and Decrementing the Pointer**

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

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Inputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 I+</td>
<td>Increments ( iii ) pointer by one</td>
<td>none</td>
</tr>
<tr>
<td>6 I-</td>
<td>Decrements ( iii ) pointer by one</td>
<td>none</td>
</tr>
<tr>
<td>7 J+</td>
<td>Increments ( jjj ) pointer by one</td>
<td>none</td>
</tr>
<tr>
<td>8 J-</td>
<td>Decrements ( jjj ) pointer by one</td>
<td>none</td>
</tr>
</tbody>
</table>

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

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

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Inputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>9 MS</td>
<td>Stores value in X-reg into current element</td>
<td>Value in X-Reg</td>
</tr>
<tr>
<td>10 MR</td>
<td>Recalls current element to X-reg</td>
<td>None. Returns element to X-reg</td>
</tr>
<tr>
<td>11 MSC+</td>
<td>Stores value in X-reg to current element and advances pointer to ( next ) element in ( column )</td>
<td>Value in X-reg.</td>
</tr>
<tr>
<td>12 MSR+</td>
<td>Stores value in X-reg to current element and advances pointer to ( next ) element in ( row )</td>
<td>Value in X-reg.</td>
</tr>
<tr>
<td>13 MRC+</td>
<td>Recalls current element to X-reg and then advances pointer to ( next ) element in ( column )</td>
<td>None. Returns element value to X-reg</td>
</tr>
<tr>
<td>14 MRR+</td>
<td>Recalls current element to X-reg and then advances pointer to ( next ) element in ( row )</td>
<td>None. Returns element value to X-reg</td>
</tr>
<tr>
<td>15 MRC-</td>
<td>Recalls current element to X-reg and then decrements pointer to ( previous ) in ( column )</td>
<td>None. Returns element value to X-reg</td>
</tr>
<tr>
<td>16 MRR-</td>
<td>Recalls current element to X-reg and then decrements pointer to ( previous ) one in ( row ).</td>
<td>None. Returns element value to X-reg</td>
</tr>
</tbody>
</table>
When the end of a column or row is reached, the pointer's index advances to the next (or previous) column or row. If the pointer's index is moved beyond the boundaries of the matrix, it cannot be moved back using these functions. You must use \texttt{MSIJ} or \texttt{MSIJA}.

The following sequence of keystrokes will create the matrix \texttt{ABC} (in extended memory).

\[
\begin{bmatrix}
5 & 6 & 7 \\
8 & 9 & 10
\end{bmatrix}
\]

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

\textbf{Updated Matrix Editor: Row Input mode.}

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

- The maximum length for all values and the blank spaces in between them is 24 characters, as it uses the Alpha register to temporarily hold them.

- Decimal and negative values are supported in this mode, but values with exponential notation (i.e. 2.4 \texttt{E23}) cannot be entered using \texttt{PMTM}.

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

<table>
<thead>
<tr>
<th>Keystrokes</th>
<th>Display</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{ALPHA, &quot;ABC&quot;, ALPHA}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.003, \texttt{XEQ &quot;MATDIM&quot;}</td>
<td>2.003</td>
<td>Dimensions matrix ABC in X-Mem.</td>
</tr>
<tr>
<td>\texttt{XEQ &quot;PMTM&quot;}</td>
<td>&quot;R1:=&quot;</td>
<td>prompts to enter the first row</td>
</tr>
<tr>
<td>5, ENTER(^\uparrow), 6, ENTER(^\uparrow), 7, R/S</td>
<td>&quot;R2:=&quot;</td>
<td>prompts for the second row</td>
</tr>
<tr>
<td>8, ENTER(^\uparrow), 9, ENTER(^\uparrow), 10, R/S</td>
<td>done!</td>
<td></td>
</tr>
</tbody>
</table>
This section briefly defines the matrix functions besides the dimensioning, storing, and recalling functions discussed above. Note that most of these functions are not meaningful for matrices containing Alpha data and that many of these functions are not meaningful for complex matrices. In any case, a complex matrix appears as a real matrix to all functions except CMEDIT. Refer to “Working with Complex Matrices” for more information on using these functions with complex matrices.

2.3.1. Matrix Arithmetic

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

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

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

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

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

\[
\begin{align*}
c_{ij} &= a_{ij} + x \quad (\text{for all } i, j \text{ in } C) \\
c_{ij} &= x + b_{ij} \\
c_{ij} &= a_{ij} + b_{ij} \\
\end{align*}
\]

\[
\begin{align*}
c_{ij} &= a_{ij} - x \quad (\text{for all } i, j \text{ in } C) \\
c_{ij} &= x - b_{ij} \\
c_{ij} &= a_{ij} - b_{ij} \\
\end{align*}
\]

\[
\begin{align*}
c_{ij} &= a_{ij} \times x \quad (\text{for all } i, j \text{ in } C) \\
c_{ij} &= x \times b_{ij} \\
c_{ij} &= a_{ij} \times b_{ij} \\
\end{align*}
\]

\[
\begin{align*}
c_{ij} &= a_{ij} \div x \quad (\text{for all } i, j \text{ in } C) \\
c_{ij} &= x \div b_{ij} \\
c_{ij} &= a_{ij} \div b_{ij} \\
\end{align*}
\]

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

\[
c_{ij} = \sum_{k=1}^{p} a_{ik} \times b_{kj}, \quad \text{where } A \text{ has } p \text{ columns and } B \text{ has } p \text{ rows.}
\]
2.3.2. Major Matrix Operations.

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

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>MDET</td>
<td>Finds the Determinant of a square matrix</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Matrix Name in Alpha</td>
</tr>
<tr>
<td>2</td>
<td>MINV</td>
<td>Inverts and replaces the square matrix</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Matrix Name in Alpha</td>
</tr>
<tr>
<td>3</td>
<td>MSYS</td>
<td>Solves a system of linear equations</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Matrix Name A. Name B in Alpha</td>
</tr>
<tr>
<td>4</td>
<td>MTRPS</td>
<td>Transposes and replaces the real matrix</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Matrix name in Alpha</td>
</tr>
</tbody>
</table>

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

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

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

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

**Example 1.**

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

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

```
ALPHA, "R0", ALPHA  
3.003, XEQ "MATDI M"
```

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

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

And now the festival begins - type:

```
XEQ "TRNPS",          R0 is transposed  
XEQ "MINV",          R0 (which was transposed) is inverted  
XEQ "MDET"           -> 0.040 is the solution.
```
Note that if you had wanted to find the transpose of the original matrix after having found its determinant, you would have needed to invert the matrix twice to change the LU-decomposed form back to the original matrix.

**LU-Decomposition**

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

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

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

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

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

- LU-decomposition destroys the original form of the matrix. So if you perform **MSYS** or **MDET** and then try to look at your input matrix (**A** in the **MATRX** program), you will find only the altered, decomposed form.

- You cannot calculate the transpose (**TRNPS**; **[SHIFT]**[B] in **MATRX** program) of a matrix in LU-decomposed form. LU-decomposition does not hinder the correct calculation of the inverse, determinant, or solution matrix, since these operations require the LU-decomposition anyway.

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

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

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

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

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

- Set flag 08 before executing **MEDIT** or **CMEDIT**. This allows you to view the elements without modifying them.
Header Register X-ray. \{ **LU?** \}

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

```
   13  12  11  10   9   8   7   6   5   4   3   2   1   0

"4" File Addr  LU?  # of Columns  Active ij  File Size
```

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

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

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

Working with Complex Matrices.

When working with complex matrices it is most important to remember that, in the calculator, a complex matrix is simply a real matrix with four times as many elements. Only the **MATRX** program and the complex-matrix editor (**CMEDIT**) “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

\[
\begin{pmatrix}
100 & -200 \\
200 & 100
\end{pmatrix}
\]

The 2 x 1 complex matrix

\[
\begin{pmatrix}
1 + 2i \\
3 - 4i
\end{pmatrix}
\]

is stored as

\[
\begin{pmatrix}
1 & -2 \\
2 & 1 \\
3 & 4 \\
-4 & 3
\end{pmatrix}
\]

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

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

In addition, a complex result-matrix from the **MATRX** program cannot be used for many complex-matrix operations outside of **MATRX**. This is because **MATRX** will dimension a complex column matrix differently than 2m x 2. Instead, it uses the dimensions 2m X 1, in which the real and imaginary parts of a number become successive elements in a single column.
This form has the advantage of saving memory and speeding up operations. The complex-matrix editor and MSYS can also use this 2m X 1 form, though they do not require it. This means you can use MSYS on a matrix system from MATRX. You can convert an existing 2m x 2 complex column matrix to the 2m X 1 form by transposing it, redimensioning it to 1 x 2m, then retransposing it. There is no easy way back.

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

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

**Using Functions with Complex Matrices**

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

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

Both the input and result matrices must be complex.

**Example 2.**

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

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

\[
\begin{bmatrix}
10 + 200i & -200i \\
-200i & (200 - 30)i
\end{bmatrix}
\begin{bmatrix}
1 \\
2
\end{bmatrix}
= \begin{bmatrix} 5 \\
0 \end{bmatrix}
\]

We'll use the individual matrix functions instead of MATRX program, already covered in the previous sections.
The main thing to sort out in this example is the dimension of the matrices involved. The coefficients matrix \( A \) is a 2 \( \times \) 2 complex matrix, thus as per the previous paragraphs we will need \((4 \times 4 + 1) = 17\) registers. The independent terms matrix \( B \) is a 2 \( \times \) 1 complex matrix, thus will need \((4 \times 2 + 1) = 9\) registers.

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

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

\[ \text{ALPHA, "R", ALPHA} \quad \text{4.004, XEQ "MATDIM"} \quad \text{ALPHA, "R17", ALPHA} \quad \text{4.002, XEQ "MATDIM"} \]

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

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

\[ \text{Keystrokes} \quad \text{Display} \]
\[ \text{ALPHA, "R", R17, ALPHA} \quad \text{ALPHA, "R17", ALPHA} \]
\[ \text{XEQ "MSYS"} \quad \text{0.0000} \]
\[ \text{SP 08} \]
\[ \text{ALPHA, R17, ALPHA} \]
\[ \text{XEQ "CMEDIT"} \]
\[ \text{R/S} \quad \text{R/S} \quad \text{R/S} \]
\[ \text{RE.1:1 = ?} \quad \text{RE.1:1 = 0.0372} \]
\[ \text{RE.2:1 = ?} \quad \text{RE.2:1 = 0.0437} \]
\[ \text{IM.1:1 = 0.1311} \quad \text{IM.2:1 = 0.1543} \]
\[ \text{IM.2:1 = 0.1543} \]

\text{(c) Ángel Martin - August 2013}
The solution is:

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

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

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

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

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

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

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

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

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

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

**Example 3.**

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

<table>
<thead>
<tr>
<th></th>
<th>Week-1</th>
<th>Week-2</th>
<th>Week-3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Combined Weight</td>
<td>274</td>
<td>233</td>
<td>331</td>
</tr>
<tr>
<td>Combined Value</td>
<td>$130.32</td>
<td>$112.96</td>
<td>$151.36</td>
</tr>
</tbody>
</table>

The following set of linear equations describes the two unknowns (the weights of cabbage and broccoli) for all three weeks, where the first row of the constant matrix represents the weights of cabbage for the three weeks and the second row represents the weights of broccoli. Since the constant matrix is not a column matrix, you must use MSYS and not the SE function in the MATRIX program.
Where the subindices indicate the crop (1= broccoli, 2=cabbage), and the week (1,2,3), and the first row describes the weight equations, and the second the prices relationship.

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

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

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

With "FACTORS" in Alpha we type:

\text{XEQ} "\text{PMTM}" -> at the prompt "R1: _" we type: 1, ENTER^, 1, R/S

\text{-> at the prompt "R2: _" we type: 0, [,], 2, 4, ENTER^, 0, [,], 8, 6, R/S}

With "LINKS" in Alpha we type:

\text{XEQ} "\text{PMTM}" -> at the prompt "R1: _" we type: 2,7,4, ENTER^, 2,3,3, ENTER^, 3,3,3, R/S

\text{-> at the prompt "R2: _" we type: 1,2,0,[,],3,2, ENTER^, 1,1,2,[,],9,6, ENTER^, 1,5,1,[,],3,6, R/S}

All set up we simply execute \text{MSYS} to obtain the solutions shought for:

\text{XEQ} "\text{MSYS}"

<table>
<thead>
<tr>
<th></th>
<th>Week-1</th>
<th>Week-2</th>
<th>Week-3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cabbage Weight (kg)</td>
<td>186</td>
<td>141</td>
<td>215</td>
</tr>
<tr>
<td>Broccoli Weight (kg)</td>
<td>88</td>
<td>92</td>
<td>116</td>
</tr>
</tbody>
</table>

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

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

\text{OMR} and \text{OMC} are extension functions – pretty much like \text{PMTM} is - and will be described in detail in chapter 3.
2.3.3.- Other Matrix Functions (“Utilities”)

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

Moving and Exchanging Matrix Sections.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 C&lt;&gt;C</td>
<td>Exchange columns k and l in a matrix</td>
<td>Name in Alpha, kkk.lll in X-reg</td>
</tr>
<tr>
<td>2 R&lt;&gt;R</td>
<td>Exchange Rows k and l in a matrix</td>
<td>Name in Alpha, kkk.lll in X-reg</td>
</tr>
<tr>
<td>3 MMOVE</td>
<td>Matrix Move</td>
<td>Names in Alpha, Pointers in stack</td>
</tr>
<tr>
<td>4 MSWAP</td>
<td>Matrix Swap</td>
<td>Names in Alpha, Pointers in stack</td>
</tr>
</tbody>
</table>

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

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

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

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

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

For example in a 4 x 5 matrix:

<table>
<thead>
<tr>
<th>Y-Register</th>
<th>Pointer Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.000</td>
<td>4.005</td>
</tr>
<tr>
<td>3.000</td>
<td>3.005</td>
</tr>
<tr>
<td>0.003</td>
<td>4.003</td>
</tr>
</tbody>
</table>
## Miscellaneous Arithmetic Functions: Maxima and Minima

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Input / Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAX</td>
<td>Finds the maximum element in matrix. Sets element pointer to it.</td>
<td>Matrix Name in Alpha. Outputs element value to X-reg</td>
</tr>
<tr>
<td>MIN</td>
<td>Finds the minimum element in matrix. Sets element pointer to it.</td>
<td>Matrix Name in Alpha. Outputs element value to X-reg</td>
</tr>
<tr>
<td>MAXAB</td>
<td>Like MAX but in absolute value. Sets element point to it.</td>
<td>Matrix Name in Alpha. Outputs element value to X-reg</td>
</tr>
<tr>
<td>CMAXAB</td>
<td>Finds maximum absolute value in k-th column. Sets element pointer to it.</td>
<td>Matrix name in Alpha, kkk in X-reg. Outputs element value to X-reg</td>
</tr>
<tr>
<td>RMAXAB</td>
<td>Finds maximum absolute value in k-th row. Sets element pointer to it.</td>
<td>Matrix name in Alpha, kkk in X-reg. Outputs element value to X-reg</td>
</tr>
</tbody>
</table>

## Miscellaneous Arithmetic functions: Norms and Sums

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Input / Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>CNRM</td>
<td>Column Norm. Finds the largest sum of the absolute values of the elements in each column of matrix.</td>
<td>Matrix name in Alpha. Outputs column norm to X-reg. Sets pointer to first element of column.</td>
</tr>
<tr>
<td>FNRM</td>
<td>Frobenius Norm. Calculates the square root of the sum of the squares of all elements in matrix.</td>
<td>Matrix name in Alpha. Outputs frobenius norm into X-reg</td>
</tr>
<tr>
<td>RNRM</td>
<td>Row Norm. Finds the largest sum of the absolute values of the elements in each row of matrix.</td>
<td>Matrix name in Alpha. Outputs row norm to X-reg. Sets pointer to first element of row.</td>
</tr>
<tr>
<td>SUM</td>
<td>Sums all elements in matrix.</td>
<td>Matrix name in Alpha. Outputs the sum to X-reg</td>
</tr>
<tr>
<td>SUMAB</td>
<td>Sums absolute values of all elements in matrix.</td>
<td>Matrix name in Alpha. Outputs the sum to X-reg</td>
</tr>
<tr>
<td>CSUM</td>
<td>Finds the sum of each column and stores them in a result vector.</td>
<td>Matrix name, result matrix name (Vector) in Alpha. (*)</td>
</tr>
<tr>
<td>RSUM</td>
<td>Finds the sum of each row and stores the sums in a result vector.</td>
<td>Matrix name, result matrix name (Vector) in Alpha. (*)</td>
</tr>
</tbody>
</table>

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

## Miscellaneous Arithmetic functions: Matrix Reductions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Input / Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>YC+C</td>
<td>Multiplies each element in column k of matrix by value in Y-ref. and adds it to corresponding element in column l.</td>
<td>Matrix name in Alpha, kkk.iii in X-reg, y in Y-reg. It changes the elements in column l</td>
</tr>
<tr>
<td>PIV</td>
<td>Finds the pivot value in column k, that is the maximum absolute value of an element on or below the diagonal.</td>
<td>Matrix Name in Alpha, kkk in X-reg</td>
</tr>
<tr>
<td>R&gt;R?</td>
<td>Compares elements in rows k and l. If (and only if) the first non-equal element in k is greater than its corresponding element in l, then the comparison is positive for the “do if true” rule of programming.</td>
<td>Matrix name in Alpha, kkk.iii in X-reg. Outputs “YES” if first non-equal element in row k is greater than element in row l. “NO” in all other case.</td>
</tr>
</tbody>
</table>
The last two functions are not operating on a matrix, but are auxiliary for the FOCAL programs:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Input / Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>20 AIP</td>
<td>Appends the absolute value of the integer part of the number in X to the contents of the Alpha register.</td>
<td>Value in X.</td>
</tr>
<tr>
<td>21 MPT</td>
<td>Appends a matrix prompt “rrr.ccc=” to the contents of the Alpha register (dropping leading zeros in each part)</td>
<td>rrr.ccc in X-reg</td>
</tr>
</tbody>
</table>

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

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

**Example.** Calculate the Row, Column and Frobenius norms for the matrix

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

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

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

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

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

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

3.1. The Enhanced Matrix Editor(s)

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

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Input / Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>PMTM</td>
<td>Prompt Matrix by Rows</td>
</tr>
<tr>
<td>2</td>
<td>IMR</td>
<td>Input Matrix by Rows</td>
</tr>
<tr>
<td>3</td>
<td>IMC</td>
<td>Input Matrix by Columns</td>
</tr>
<tr>
<td>4</td>
<td>OMR</td>
<td>Output Matrix by Rows</td>
</tr>
<tr>
<td>5</td>
<td>OMC</td>
<td>Output Matrix by Column</td>
</tr>
<tr>
<td>6</td>
<td>OXC</td>
<td>Output Column k</td>
</tr>
<tr>
<td>7</td>
<td>OXR</td>
<td>Output Row k</td>
</tr>
</tbody>
</table>

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

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

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

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

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

The two limitations of PMTM are as follows:

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

Obviously this makes PMTM the ideal choice for matrices containing integer numbers as elements – but not exclusively so as it can also be used for other values (real-numbers) as long as the two conditions above are respected.
At the heart of PMTM there is is function ^MROW (“Enter Matrix Row”), responsible for the presentation of the prompt in Alpha and accepting the keyboard pressings there to make up the string (or list) with all values. It also provides the logic of actions for the control keys, like ENTER^, Back arrow, R/S, etc.

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

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

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

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

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

```
R 1: 1 3 20
   USER 1 3 PRM
```

and

```
R 2: 4 3 -.28
   USER 1 3 PRM
```

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

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

The remaining routines in this section all deal with Input and Output of the matrix elements, depending on whether it’s done following the Row or Column sequence, as well as two functions to only view one specific row or column (OXR and OXC).
They are very much equivalent to **MEDIT** in many aspects, although the symbol “a” is used in the prompts. They are slightly faster and offer the added convenient feature that for integer element **values** the zeros after the decimal point are not shown in the prompt – regardless of the current display settings (FIX or otherwise). This makes for a clearer UI.

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

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

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

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

3.3.1. Utility / housekeeping functions: rounding the capabilities.

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

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

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

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

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

<table>
<thead>
<tr>
<th>1 LBL &quot;MAT=&quot;</th>
<th>&quot;A,B&quot; expected in Alpha</th>
<th>1 LBL &quot;MCON&quot;</th>
<th>MNAME in Alpha</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 DIM?</td>
<td>dimension</td>
<td>2 MZERO</td>
<td>clear all elements</td>
</tr>
<tr>
<td>3 ASWAP</td>
<td>swap Alpha</td>
<td>3 RDN</td>
<td>get constant back to X</td>
</tr>
<tr>
<td>4 MATDIM</td>
<td>re-dimension target</td>
<td>4 &quot;X&quot;</td>
<td>prepare alpha string</td>
</tr>
<tr>
<td>5 ASWAP</td>
<td>undo the swap</td>
<td>5 MAT+</td>
<td>add x to all elements</td>
</tr>
<tr>
<td>6 CLST</td>
<td>prepare pointers</td>
<td>6 MNAME?</td>
<td>recall MNAME to Alpha</td>
</tr>
<tr>
<td>7 MMOVE</td>
<td>move all elements</td>
<td>7 END</td>
<td>done</td>
</tr>
<tr>
<td>8 END</td>
<td>done</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**MZERO** is the unsung hero behind other routines – as the proper way to clear a matrix file, since **CLFL** cannot be used because it also clears the header register (it was meant for Data files). Use it safely for matrices in main and x-memory.

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

<table>
<thead>
<tr>
<th>1</th>
<th>LBL &quot;MSORT&quot;</th>
<th>2</th>
<th>LBL &quot;MZERO&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>SIZE?</td>
<td>2</td>
<td>DIM?</td>
</tr>
<tr>
<td>3</td>
<td>MSZE?</td>
<td>3</td>
<td>SF 25</td>
</tr>
<tr>
<td>4</td>
<td>E</td>
<td>4</td>
<td>PURFL</td>
</tr>
<tr>
<td>5</td>
<td>X-Y?</td>
<td>5</td>
<td>FC?C 25</td>
</tr>
<tr>
<td>6</td>
<td>FSIZE</td>
<td>6</td>
<td>GTO 01</td>
</tr>
<tr>
<td>7</td>
<td>&quot;-F&quot;</td>
<td>7</td>
<td>MATDIM</td>
</tr>
<tr>
<td>8</td>
<td>MAT=</td>
<td>8</td>
<td>RTN</td>
</tr>
<tr>
<td>9</td>
<td>MSZE?</td>
<td>9</td>
<td>LBL 01</td>
</tr>
<tr>
<td>10</td>
<td>its size again</td>
<td>10</td>
<td>ANUM</td>
</tr>
<tr>
<td>11</td>
<td>E-3/1+</td>
<td>11</td>
<td>ENTER^</td>
</tr>
<tr>
<td>12</td>
<td>RGSORT</td>
<td>12</td>
<td>MSZE?</td>
</tr>
<tr>
<td>13</td>
<td>ASWAP</td>
<td>13</td>
<td>+</td>
</tr>
<tr>
<td>14</td>
<td>CST</td>
<td>14</td>
<td>E3/3+</td>
</tr>
<tr>
<td>15</td>
<td>MMOVE</td>
<td>15</td>
<td>+</td>
</tr>
<tr>
<td>16</td>
<td>SNUMF?</td>
<td>16</td>
<td>CLRGX</td>
</tr>
<tr>
<td>17</td>
<td>END</td>
<td>17</td>
<td>END</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>1</th>
<th>MSZE?</th>
<th>2</th>
<th>A60A</th>
<th>3</th>
<th>A60B</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>MSZE?</td>
<td>2</td>
<td>0BF</td>
<td>3</td>
<td>005</td>
</tr>
<tr>
<td>3</td>
<td>MSZE?</td>
<td>2</td>
<td>&quot;?&quot;</td>
<td>3</td>
<td>&quot;E&quot;</td>
</tr>
<tr>
<td>4</td>
<td>MSZE?</td>
<td>2</td>
<td>&quot;D&quot;</td>
<td>3</td>
<td>&quot;Z&quot;</td>
</tr>
<tr>
<td>5</td>
<td>MSZE?</td>
<td>2</td>
<td>&quot;S&quot;</td>
<td>3</td>
<td>&quot;M&quot;</td>
</tr>
<tr>
<td>6</td>
<td>MSZE?</td>
<td>A60F</td>
<td>179</td>
<td>PORT DEP:</td>
<td>Jumps to Bank_2</td>
</tr>
<tr>
<td>7</td>
<td>MSZE?</td>
<td>A610</td>
<td>03C</td>
<td>XQ</td>
<td>adds &quot;4&quot; to [XS]</td>
</tr>
<tr>
<td>8</td>
<td>A611</td>
<td>109</td>
<td>--&gt;ASD9</td>
<td>[LNCH0]</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>valid for main and X-mem</td>
<td>A612</td>
<td>388</td>
<td>&lt;parameter&gt;</td>
<td>B78B</td>
</tr>
<tr>
<td>10</td>
<td>the proper way to do it</td>
<td>A613</td>
<td>00B</td>
<td>JNC +01</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>A614</td>
<td>100</td>
<td>ENROM1</td>
<td>restore bank-1</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>MSZE?</td>
<td>A615</td>
<td>080</td>
<td>C=N ALL</td>
<td>header register</td>
</tr>
<tr>
<td>13</td>
<td>MSZE?</td>
<td>A616</td>
<td>106</td>
<td>A=C S&amp;X</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>MSZE?</td>
<td>A618</td>
<td>0CD</td>
<td>--&gt;315F</td>
<td>[ATOX20]</td>
</tr>
</tbody>
</table>

**PMAT** is nothing more than a user-friendly driver program to automate the complete matrix product procedure, without any need to dimension the result matrix in advance. The routine will guide you step-by-step thru the complete sequence, including the element data input and output.
**MIDN** is a good example of a sorely missing function: the majority of matrix algorithms involve identity matrices, one way or another, so having a routine that does the job becomes rather important. The SandMatrix routine follows a single-element approach, storing ones in the main diagonal after zeroing the matrix first. This is faster and more convenient than block-based methods, even if not requiring scratch matrices for intermediate calculations. See an example below courtesy of Thomas Klemm:

\[
\begin{bmatrix}
1 & 1 & 1 \\
1 & 0 & 0 \\
0 & 0 & 0
\end{bmatrix} \rightarrow 
\begin{bmatrix}
1 & 0 & 0 \\
0 & 0 & 0 \\
0 & 0 & 0
\end{bmatrix} \rightarrow 
\begin{bmatrix}
1 & 0 & 0 \\
0 & 1 & 0 \\
0 & 0 & 1
\end{bmatrix}
\]

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

<table>
<thead>
<tr>
<th>1</th>
<th>LBL &quot;MRDIM&quot;</th>
<th>MNAME in Alpha</th>
<th>16</th>
<th>X&lt;&gt;Y</th>
<th>min(1,2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>DIM?</td>
<td></td>
<td>17</td>
<td>RCL Z</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>X&lt;&gt;Y</td>
<td>new dimension to X</td>
<td>18</td>
<td>INT</td>
<td>min(I)</td>
</tr>
<tr>
<td>4</td>
<td>ASTO T</td>
<td>temporary safekeep</td>
<td>19</td>
<td>+</td>
<td>min(I), min(J)</td>
</tr>
<tr>
<td>5</td>
<td>&quot;J-&quot;</td>
<td>add tilde</td>
<td>20</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>MATDIM</td>
<td>create new matrix</td>
<td>21</td>
<td>STO Z</td>
<td>prepare pointers</td>
</tr>
<tr>
<td>7</td>
<td>CLA</td>
<td></td>
<td>22</td>
<td>ASTO T</td>
<td>temporary safekeep</td>
</tr>
<tr>
<td>8</td>
<td>ARCL T</td>
<td>MNAME</td>
<td>23</td>
<td>&quot;J-&quot;</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>X&lt;&gt;Y</td>
<td>min(1,2)</td>
<td>24</td>
<td>ARCL T</td>
<td>MNAME</td>
</tr>
<tr>
<td>10</td>
<td>X&lt;&gt;Y</td>
<td>min(1,2)</td>
<td>25</td>
<td>&quot;J-&quot;</td>
<td>prepare Alpha string</td>
</tr>
<tr>
<td>11</td>
<td>STO Z</td>
<td>keep in Z</td>
<td>26</td>
<td>MMOVE</td>
<td>copy elements</td>
</tr>
<tr>
<td>12</td>
<td>FRC</td>
<td></td>
<td>27</td>
<td>PURFL</td>
<td>purge original file</td>
</tr>
<tr>
<td>13</td>
<td>X&lt;&gt;Y</td>
<td>MNAME?</td>
<td>28</td>
<td>MNAME?</td>
<td>recall name to Alpha</td>
</tr>
<tr>
<td>14</td>
<td>FRC</td>
<td></td>
<td>29</td>
<td>END</td>
<td>done</td>
</tr>
</tbody>
</table>

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

1. creating a new matrix file and copying it over once again, or (preferable)
2. using **RENMFL** (in the AMC_OS/X module) to rename the X-mem file
Finding an element within a Matrix $\{\text{MFIND}\}$ - plus an easy-driver for $M \times M$

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

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

Below are the program listings for your perusal.

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

Note that in $\text{MATP}$ I have chosen $\text{PMTM}$ to enter the element data values – therefore it’s somehow limited by the same constraints described before, ie. total length in Alpha and no support for the EEX key.
### 3.2.2. New Math functions.- Completing the core function set.

The next group includes advanced application areas in “core” matrix math.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Input / Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>9 M^1/X</td>
<td>Brute-force Matrix X-th Root</td>
<td>Matrix name in Alpha, order in X. The result matrix replaces the input.</td>
</tr>
<tr>
<td>10 M^2</td>
<td>Square power of a square Matrix</td>
<td>Matrix name in Alpha. The result matrix replaces the input.</td>
</tr>
<tr>
<td>11 MDPS</td>
<td>Matrix Diagonal Product Sum</td>
<td>Matrix name in Alpha. Output is result in X-reg.</td>
</tr>
<tr>
<td>12 MEXP</td>
<td>Exponential of a Matrix</td>
<td>Matrix name in Alpha. The result matrix replaces the input.</td>
</tr>
<tr>
<td>13 MLIE</td>
<td>Matrix Lie Product</td>
<td>Matrix names in Alpha: “A,B,C”. Result matrix C must be different.</td>
</tr>
<tr>
<td>14 MLN</td>
<td>Matrix Logarithm</td>
<td>Matrix name in Alpha. The result matrix replaces the input.</td>
</tr>
<tr>
<td>15 MPWR</td>
<td>Matrix Power of integer order</td>
<td>Matrix name in Alpha, order in X-reg. The result matrix replaces the input.</td>
</tr>
<tr>
<td>16 MSQRT</td>
<td>Matrix Square Root</td>
<td>Matrix name in Alpha. The result matrix replaces the input.</td>
</tr>
<tr>
<td>17 MTRACE</td>
<td>Calculates the Trace of a Square Matrix</td>
<td>Matrix name in Alpha. Output is put into W-reg.</td>
</tr>
<tr>
<td>18 R/aRR</td>
<td>Row division by diagonal element</td>
<td>Matrix name in Alpha, row kkk in X-reg. All row elements divided by akk.</td>
</tr>
<tr>
<td>19 ΣIJJI</td>
<td>Sum of crossed-elements products</td>
<td>Matrix name in Alpha. Output is put in X-reg.</td>
</tr>
</tbody>
</table>

---

**Formulae and algorithms used.**

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

### Matrix Exponential \{ MEXP \}

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

\[
e^{\mathbf{X}} = \sum_{k=0}^{\infty} \frac{1}{k!} \mathbf{X}^k
\]

where \( \mathbf{X}^0 \) is the identity matrix, \( \mathbf{I} \). The above series always converges, so the exponential of \( \mathbf{X} \) is well-defined. Note that if \( \mathbf{X} \) is a \( 1 \times 1 \) matrix the matrix exponential of \( \mathbf{X} \) is a \( 1 \times 1 \) matrix consisting of the ordinary exponential of the single element of \( \mathbf{X} \).

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

\[
\exp(\mathbf{A}) = \mathbf{I} + \mathbf{A} + \mathbf{A}^2/2! + \mathbf{A}^3/3! + ..... + \mathbf{A}^k/k! + ....
\]
The program adds new terms until their contribution is negligible, i.e. it results in the same matrix after adding it. This by itself poses an interesting question: how to check whether two matrices are the same? Obviously doing it element-to-element would be a long and impractical method. The alternative is to use the matrix Frobenius norm as a surrogate criterion; assuming that for very similar matrices, they’ll be equal when they have the same norm.

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

Logarithm of a Matrix \{ MLN \}

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

The SandMatrix uses the following algorithm:

\[
\text{If } ||A - I|| < 1, \text{ the logarithm of an } n \times n \text{ matrix } A \text{ is defined by the series expansion:}
\]

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

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

\[
A = \begin{bmatrix} 1 & 2 & 3 \\ 0 & 1 & 2 \\ 1 & 3 & 2 \end{bmatrix}
\]

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

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

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

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

\[
A = \begin{bmatrix} 1.2 & 0.1 & 0.3 \\ 0.1 & 0.8 & 0.1 \\ 0.1 & 0.2 & 0.9 \end{bmatrix}
\]

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

The result matrix after executing \([MLN]\) is as follows:

\[
\text{Ln}(A) = \begin{bmatrix} 0.167083396 & 0.069577923 & 0.287707999 \\ 0.097783005 & -0.240971674 & 0.103424021 \\ 0.086500972 & 0.235053124 & -0.131906636 \end{bmatrix}
\]
So we see that unfortunately the logarithm is not a trivial exercise. The programs are listed below, note the combination of both exponential and logarithm into a single program, with flag 01 controlling the case.

```
1 LBL "MLN" 44 LBL 02
2 SF 01 exp flag 45 VIEW 00
3 GTO 00 46 "\#".
4 LBL "MEXP" 47 ARCL 01
5 CF 01 LN flag 48 "\#\#"
6 LBL 00 49 M\*M
7 DIM? get dimension 50 "P\#".
8 IM? not square? 51 CLST
9 -ADV MATRIX error message 52 MMOVE
10 ASTO 01 53 RCL 02
11 */\#" 54 FC? 01 exp?
12 MAT= safekeeping copy 55 FACT to be used as divisor
13 DIM? get dimension 56 FC? 01 exp?
14 "\#" 57 GTO 04
15 M\*DIM auxiliary matrix 58 ENTER\^ to be used as divisor
16 */\#" 59 ENTER\^ 60 E
17 M\*DIM auxiliary matrix 61 +
18 MIDN 62 CHSYX
19 ARCL 01 63 LBL 04
20 FS? 01 LN? 64 LBL 00
21 ASWAP yes, swap names 65 "P\#".
22 */\#" 66 MAT/ divide by scalar
23 FS? 01 LN? 67 E "\#\#" prepare new string
24 MAT- 68 FC? 01 exp? to be used as divisor
25 FC? 01 exp? 69 MAT= safekeeping copy
26 MAT+ 70 E 71 "\#".
27 */\#" 72 ST+ 02 increase term index
28 FNRM initial norm 73 "\#".
29 STO 00 store in RDO 74 FNRM new frobenius norm
30 FC? 01 exp? 75 X<> 00 swap with old norm
31 CLA 76 GTO 00 recall new again
32 ARCL 01 77 X\#Y? are the different?
33 FC? 01 exp? 78 GTO 02 yes, keep it
34 GTO 04 79 ARCL 01 no, we’re done
35 MAT= 80 "P\#".
36 CLAC 81 PURFL purges "\#"
37 ABSP 82 ASWAP
38 LBL 04 83 PURFL purges "\#"
39 */\#" 84 MNAME? recalls name to Alpha
40 CLST 85 END
41 MMOVE 86 END
42 2
43 STO 02
```

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

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

As usual, these routines will result in “ALPHA DATA” if the matrix is in LU decomposed form.
Square root of a Matrix \{ \text{MSQRT} \}

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

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

Computing the matrix square root in the SandMatrix uses a modification of the Denman-Beavers iteration. Let \( Y_0 = A \) and \( Z_0 = I \), where \( I \) is the \( n \times n \) identity matrix. The iteration is defined by

\[
Y_{k+1} = \frac{1}{2} \left( Y_k + Z_k^{-1} \right), \\
Z_{k+1} = \frac{1}{2} \left( Z_k + Y_k^{-1} \right).
\]

Convergence is not guaranteed, even for matrices that do have square roots, but if the process converges, the matrix \( Y_k \) converges quadratically to a square root \( A^{1/2} \), while \( Z_k \) converges to its inverse, \( A^{-1/2} \).

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

Example 1. Find a square root of the 3\textsuperscript{rd}. order Hilbert matrix:

\[
A = \begin{bmatrix}
1 & 1/2 & 1/3 \\
1/2 & 1/3 & 1/4 \\
1/3 & 1/4 & 1/5
\end{bmatrix}
\]

We’ll use \text{IMR} to input the element values (as \text{PMTM} is not really suitable for this example). Previously we need to create the matrix, as follows:

\[
\text{ALPHA "HILB3", ALPHA 3.003, XEQ "MATDIM"}
\]

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

Final Frobenius norm = 1,238278374

\[
\text{Sqrt}(A) = \begin{bmatrix}
0.917390290 & 0.345469265 & 0.197600714 \\
0.345469265 & 0.374984280 & 0.270871020 \\
0.197600714 & 0.270871020 & 0.295943995
\end{bmatrix}
\]

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

\[
\begin{bmatrix}
0.9999999999 & 0.4999999999 & 0.3333333333 \\
0.5000000000 & 0.3333333333 & 0.2500000000 \\
0.3333333333 & 0.2499999999 & 0.2000000000
\end{bmatrix}
\]

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

(c) Ángel Martin - August 2013
Example 2. Find a square root of the 4 x 4 matrix below, and check the accuracy by squaring it back.

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

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

\[
\text{SQRT}(A) = \begin{bmatrix}
8.0000 & 6.0000 & 1.0000 & 7.0000 \\
-7.0000 & -1.0001 & -8.0000 & 3.0000 \\
-6.0000 & 6.0000 & 0.0000 & -6.0000 \\
6.0000 & 7.0000 & 7.0000 & 3.0000
\end{bmatrix}
\]

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

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

```
1  LBL "MSQRT"  30  X=YR?  are they equal>
2  DIM?  get dimension  31  SF 00  yes, flag this fact
3  IR?  is it square?  32  X=YR?  are they equal>
4  -ADV MATRIX  no, show error  33  GTO 02  yes, jump over
5  CF 00  34  CLA  no, keep at it
6  FNRM  initial norm  35  ARCL 01  prepare Alpha string
7  STO 00  store it in R00  36  "\'|\#"  invert matrix
8  ASTO 01  matrix name to R01  37  MINV  copy in auxiliary
9  RDN  dimension to X-reg  38  MAT=  undo the inversion
10  "p"  39  MINV  auxiliary matrix P
11  MATDIM  auxiliary matrix P  40  "Q,#,Q"  invert auxiliary
12  "Q"  41  MINV  auxiliary matrix Q
13  MATDIM  auxiliary matrix Q  42  MAT+  sum it to partial result
14  MIDN  43  "Q,X"  auxiliary matrix P
15  LBL 00  44  2  auxiliary matrix #
16  "Q,#"  45  MAT/  divide by scalar 2
17  MINV  46  LBL 02  
18  MAT=  auxiliary matrix #  47  "p,"  
19  CLA  48  ARCL 01  
20  ARCL 01  49  MAT=  
21  "\'|\#,#p"  50  FC? 00  were norms equal?
22  MAT+  51  GTO 00  no, next iteration
23  "p,X"  52  PURFL  purge P
24  2  53  "Q"  
25  MAT/  54  PURFL  purge Q
26  FNRM  Frobenius norm  55  "#"  
27  VIEW X  show progress  56  PURFL  purge #
28  X<> 00  swao with old norm  57  MNAME?  matrix name to Alpha
29  RCL 00  recall new one again  58  END  done
```

As usual, this routine will result in “ALPHA DATA” if the matrix is in LU decomposed form.
Matrix Integer Powers and Roots. \{ [M^2], \text{MPWR}, [M^{1/X}] \}

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

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

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

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

\[ [A]^{1/x} = \exp[1/x \cdot \ln(A)], \quad \text{which is only valid when} \quad \text{abs}([A-I]) < 1 \]

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

\textbf{Example 1.} Calculate the 7-th. power of the matrix below:

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

Type XEQ "MPWR", and the result is:

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

\textbf{Example 2.} Calculate the 5-th. root of matrix \( A \) below, then compare its 5-th power to the original matrix.

\[
A = \begin{bmatrix}
1.2 & 0.1 & 0.3 \\
0.1 & 0.8 & 0.1 \\
0.1 & 0.2 & 0.9 \\
\end{bmatrix}
\]

The results are as follows:

\[
A^{1/5} = \begin{bmatrix}
1.034632528 & 0.015156701 & 0.057916477 \\
0.019601835 & 0.953558110 & 0.020490861 \\
0.017823781 & 0.045426856 & 0.974937998 \\
\end{bmatrix}
\]

\[
[A^{1/5}]^5 = \begin{bmatrix}
1.199999994 & 0.100000000 & 0.300000000 \\
0.100000000 & 0.800000000 & 0.100000000 \\
0.100000000 & 0.200000000 & 0.900000000 \\
\end{bmatrix}
\]
Program listings for MPWR, M^2 and M^1/X.

<table>
<thead>
<tr>
<th>1</th>
<th>LBL &quot;MPWR&quot;</th>
<th>MNAME in Alpha</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>DIM?</td>
<td>get dimension</td>
</tr>
<tr>
<td>3</td>
<td>IM?</td>
<td>square?</td>
</tr>
<tr>
<td>4</td>
<td>-ADV MATRIX</td>
<td>yes, show error</td>
</tr>
<tr>
<td>5</td>
<td>-CCD MATRIX</td>
<td>no, show &quot;RUNNING...&quot;</td>
</tr>
<tr>
<td>6</td>
<td>X=Y</td>
<td>power index to X-reg</td>
</tr>
<tr>
<td>7</td>
<td>INT</td>
<td>make integer</td>
</tr>
<tr>
<td>8</td>
<td>X&lt;0?</td>
<td>is it negative?</td>
</tr>
<tr>
<td>9</td>
<td>MIDN</td>
<td>yes, make identity</td>
</tr>
<tr>
<td>10</td>
<td>RTN</td>
<td>done.</td>
</tr>
<tr>
<td>11</td>
<td>LBL 01</td>
<td>←</td>
</tr>
<tr>
<td>12</td>
<td>X&lt;0?</td>
<td>is it negative?</td>
</tr>
<tr>
<td>13</td>
<td>MINV</td>
<td>yes, invert matrix</td>
</tr>
<tr>
<td>14</td>
<td>ABS</td>
<td>15</td>
</tr>
<tr>
<td>15</td>
<td>E</td>
<td>16</td>
</tr>
<tr>
<td>16</td>
<td>-</td>
<td>n-1</td>
</tr>
<tr>
<td>17</td>
<td>X&lt;0?</td>
<td>was n=1?</td>
</tr>
<tr>
<td>18</td>
<td>RTN</td>
<td>yes, we're done</td>
</tr>
<tr>
<td>19</td>
<td>STO 00</td>
<td>store in R00</td>
</tr>
<tr>
<td>20</td>
<td>ASTO 01</td>
<td>store Mname in R01</td>
</tr>
<tr>
<td>21</td>
<td>ASTO 01</td>
<td>store Mname in R01</td>
</tr>
<tr>
<td>22</td>
<td>&quot;</td>
<td>-&quot;#&quot;</td>
</tr>
<tr>
<td>23</td>
<td>MAT=</td>
<td>copy to aux matrix #</td>
</tr>
<tr>
<td>24</td>
<td>DIM?</td>
<td>get dimension</td>
</tr>
<tr>
<td>25</td>
<td>&quot;P&quot;</td>
<td>26</td>
</tr>
<tr>
<td>27</td>
<td>LBL 00</td>
<td>←</td>
</tr>
<tr>
<td>28</td>
<td>&quot;#&quot;</td>
<td>&quot;#,&quot;</td>
</tr>
<tr>
<td>29</td>
<td>ARCL 01</td>
<td>&quot;#,#NAME&quot;</td>
</tr>
<tr>
<td>30</td>
<td>&quot;</td>
<td>-P&quot;</td>
</tr>
<tr>
<td>31</td>
<td>M*M</td>
<td>matrix product</td>
</tr>
<tr>
<td>32</td>
<td>VIEW 00</td>
<td>show current index</td>
</tr>
<tr>
<td>33</td>
<td>&quot;P #&quot;</td>
<td>7</td>
</tr>
<tr>
<td>34</td>
<td>CLST</td>
<td>8</td>
</tr>
<tr>
<td>35</td>
<td>MMOVE</td>
<td>copy result to #</td>
</tr>
<tr>
<td>36</td>
<td>DSE 00</td>
<td>decrement index</td>
</tr>
<tr>
<td>37</td>
<td>GTO 00</td>
<td>loop back if not ready</td>
</tr>
<tr>
<td>38</td>
<td>&quot;#,&quot;</td>
<td>&quot;#,&quot;</td>
</tr>
<tr>
<td>39</td>
<td>ARCL 01</td>
<td>&quot;#,#NAME&quot;</td>
</tr>
<tr>
<td>40</td>
<td>MAT=</td>
<td>copy result to #</td>
</tr>
<tr>
<td>41</td>
<td>PURFL</td>
<td>purge #</td>
</tr>
<tr>
<td>42</td>
<td>&quot;P&quot;</td>
<td>43</td>
</tr>
<tr>
<td>44</td>
<td>PURFL</td>
<td>purge P</td>
</tr>
<tr>
<td>45</td>
<td>MNAME?</td>
<td>recall MNAME to Alpha</td>
</tr>
<tr>
<td>46</td>
<td>END</td>
<td>done.</td>
</tr>
</tbody>
</table>

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

An alternative listing for M^1/X that includes a convergency check is shown in next page. Note how the calculations to check for the condition are a taxiing step, in that it requires a scratch matrix to calculate its norm. On the positive side though, it'll spare us the wait for a non-convergent process that would take much longer until it's apparent so. So after some consideration the longer version is now in the module.
The scratch matrix is removed in case there is divergence, or reused to calculate the logarithm if not – thus at least it’s not all a waste of time. If there is no convergence you may still go ahead and hit R/S after the error message to see how the precision factor keeps increasing until the “OUT OF RANGE” condition.

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

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

\[
M_0 = A \quad M_{k+1} = M_k \left[ (2I + (p-2)M_k)(I + (p-1)M_k)^{-1} \right]^p \\
X_0 = I \quad X_{k+1} = X_k \left(2I + (p-2)M_k \right)^{-1}(I + (p-1)M_k)
\]

where $I$ is the Identity matrix

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

The convergence is also quadratic if $A$ has no negative real eigenvalue.
Lie Product of two Matrices.  \{ MLIE \}

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

\[
\text{Lie}(A,B) = - \text{Lie}(B,A) = AB - BA
\]

Example.- Calculate the lie product for matrices:

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

The results are:

\[
\text{ALPHA}, \ "A,B,C", \ \text{ALPHA} \quad \rightarrow \quad \text{Lie}(A,B) = \begin{bmatrix}
24 & 19 & -65 \\
58 & 85 & -34
\end{bmatrix}
\]

\[
\text{ALPHA}, \ "B,A,C", \ \text{ALPHA} \quad \rightarrow \quad \text{Lie}(B,A) = \begin{bmatrix}
-24 & -19 & 65 \\
-58 & -85 & 34
\end{bmatrix}
\]

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

In fact \texttt{SWAP} swaps the contents of the Alpha register around the \textit{first comma} character encountered, which makes it so interesting in this case.

\begin{verbatim}
1 LBL “MLIE”  \"OP1, OP2, RES\" in Alpha
2 XEQ 00  \text{calculate \texttt{[OP1][OP2]} \texttt{[OP2][OP1]}}
3 ST<>A  \text{complete string to stack}
4 MNAME?  \text{RES}
5 ASTO 00  \text{MNAME?}
6 ST<>A  \text{restores complete string}
7 CLAC  \text{\"OP1,OP2,\"}
8 ABSP  \text{\"OP1,OP2\"}
9 ASWAP  \text{\"OP2,OP1\"}
10 “;\ #:”  \text{\"OP1,OP2,\#\"}
11 XEQ 00  \text{\texttt{[OP2][OP1]}}
12 CLA  \text{\texttt{MATDIM}}
13 ARCL 00  \text{\"RES\"}
14 “;\ #:”  \text{\"RES,\#\"}
15 ARCL 00  \text{\"RES,#,RES\"}
16 MAT-  \text{\texttt{M*M}}
17 “;\ #:”  \text{\texttt{M+M}}
18 PURFL  \text{\texttt{M+M}}
19 MNAME?  \text{purge #}
20 RTN  \text{\texttt{M+M}}
21 LBL 00  \text{\texttt{M+M}}
22 DIMP  \text{\texttt{M+M}}
23 ASWAP  \text{\texttt{M+M}}
24 ASWAP  \text{\texttt{M+M}}
25 ASWAP  \text{\texttt{M+M}}
26 ASWAP  \text{\texttt{M+M}}
27 “;\ #:”  \text{\texttt{M+M}}
28 END  \text{\texttt{M+M}}
\end{verbatim}
Matrix Trace and remaining functions. \( \{ \text{MTRACE} \} \)

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

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

where \( a_{ii} \) represents the entry on the \( i \)th row and \( i \)th column of \( A \). The trace of a matrix is the sum of the (complex) eigenvalues, and it is invariant with respect to a change of basis. Note that the trace is only defined for a square matrix (i.e., \( n \times n \)).

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

\[
\text{tr}(X^T Y) = \text{tr}(X Y^T) = \text{tr}(Y^T X) = \text{tr}(X X^T) = \sum_{i,j} X_{i,j} Y_{i,j}.
\]

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

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

Eigenvalues relationships.

The trace of a matrix is intricately related to its eigenvalues. In contrast with the determinant (which is the product of its eigenvalues); if \( A \) is a square \( n \)-by-\( n \) matrix with real or complex entries and if \( \lambda_1, \ldots, \lambda_n \) are the eigenvalues of \( A \) (listed according to their algebraic multiplicities), then

\[
\text{tr}(A) = \sum_i \lambda_i.
\]

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

\[
\text{det}(e^A) = e^{\text{tr}(A)}.
\]

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

\[
\begin{align*}
\text{DIM}(n, n+1) & \quad \rightarrow \quad \text{DIM}(1, n) \\
A^2 & \quad \rightarrow \quad A \\
\text{DET} & \quad \rightarrow \quad \text{DET}
\end{align*}
\]

Here’s the sweet and short SandMatrix program listing, compared side-to-side to a block-approach alternative implementation – which also requires a scratch matrix if one wishes to keep the original matrix unchanged, as well as some additional steps for Alpha housekeeping.
Note how the alternative approach function **SUM** is used, which removes the need to calculate the determinant in the last step of the sketch. Regardless, it's bigger and takes longer execution time, even without the test for square matrix condition.

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

The last function in this chapter is used to modify the values of all elements, dividing each row by its diagonal element; that is: \[ a_{ij} = a_{ij} / a_{ii}, \quad j=1,2,\ldots, n \]

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

Other two functions directly related to the eigenvalues are \( \text{MDPS} \) and \( \Sigma_{IJJI} \). They compute sums of pairs of element multiplication, either for those in the diagonal (\( a_{ii} \times a_{kk} \)); or for “crossed” (i.e. opposite) ones, (\( a_{ij} \times a_{ji} \)), with \( i \neq j \) – excluding the diagonal. 

\[-2 \times 1 - 4 \times 2 + 3 \times 0 \]

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

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

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

\[
\text{Tr}(A) = -2 + 1 - 1 = -2
\]

\[
\text{MDPS} = (-2 \times 1) - (1 \times 1) + (2 \times 1) = -1
\]

\[
Σ_{aij \; aji} = -2 \times 1 - 4 \times 2 + 3 \times 0 = -10
\]

Program listings – easy does it, element-wise.

```
1 LBL "ΣIJJI" MNAME in Alpha
2 DIM? get dimension
3 ≠? not square?
4 -ADV MATRIX error message
5 INT n
6 E
7 + n-1
8 E3/E+ 1,00(n-1)
9 CLA
10 STO M+ set pointer to 1:1
11 LBL 00
12 RCL M k,00(n-1)
13 E
14 E3/E+ 1,001
15 + (k+1),00n
16 STO N
17 LBL 01
18 RCL M k,00(n-1)
19 INT k
20 RCL N (k+1),00n
21 INT k+1
22 I<>J does E3/ for integers
23 + (k+1),00[n+k+1]
24 MSU set pointer
25 MR recall element
26 X<>Y
27 I<>J does E3/ for integers
28 MSU set pointer
29 RDN recall element
30 * multiply
31 + a00 * ann
32 ST+ O add to partial sum
33 ISG N increase row
34 GTO 01 next element in row
35 ISG M increase column
36 GTO 00 next column
37 RCL O partial sum to X-reg
38 MNAME? recall mname to Alpha
39 END done
```
Appendix.- Square root of a 2x2 Matrix.
A square root of a 2x2 matrix $M$ is another 2x2 matrix $R$ such that $M = R^2$, where $R^2$ stands for the matrix product of $R$ with itself. In many cases, such a matrix $R$ can be obtained by an explicit formula. Let

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

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

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

Then, if $t \neq 0$, a square root of $M$ is:

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

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

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

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

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

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

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

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Input / Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHRPOL</td>
<td>Characteristic Polynomial</td>
<td>Under prgm control</td>
</tr>
<tr>
<td>EIGEN</td>
<td>Eigen Values by SOLVE</td>
<td>Under prgm control</td>
</tr>
<tr>
<td>#EV</td>
<td>Subroutine for EIGEN</td>
<td>Under prgm control</td>
</tr>
<tr>
<td>EV3</td>
<td>Eigen values 3x3</td>
<td>Matrix in X-Mem</td>
</tr>
<tr>
<td>EV3X3</td>
<td>Eigen values 3x3</td>
<td>Prompts Matrix Elements</td>
</tr>
<tr>
<td>JACOBI</td>
<td>Symmetrical Eigenvalues</td>
<td>Under prgm control</td>
</tr>
</tbody>
</table>

4.1. Eigenvectors and Eigenvalues.

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

\[
Av = \lambda v
\]

The number \( \lambda \) is called the eigenvalue of \( A \) corresponding to \( v \).

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

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

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

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

\[
\begin{bmatrix}
v_1 \\
v_2 \\
\vdots \\
v_n
\end{bmatrix}
\text{is mapped to}
\begin{bmatrix}
w_1 \\
w_2 \\
\vdots \\
w_n
\end{bmatrix}
=
\begin{bmatrix}
A_{1,1} & A_{1,2} & \cdots & A_{1,n} \\
A_{2,1} & A_{2,2} & \cdots & A_{2,n} \\
\vdots & \vdots & \ddots & \vdots \\
A_{n,1} & A_{n,2} & \cdots & A_{n,n}
\end{bmatrix}
\begin{bmatrix}
v_1 \\
v_2 \\
\vdots \\
v_n
\end{bmatrix}
\]
where, for each index $i$,

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

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

In particular, multiplication by a $3\times3$ matrix $A$ may change both the direction and the magnitude of an arrow $v$ in three-dimensional space. However, if $v$ is an eigenvector of $A$ with eigenvalue $\lambda$, the operation may only change its length, and either keep its direction or flip it (make the arrow point in the exact opposite direction). Specifically, the length of the arrow will increase if $|\lambda| > 1$, remain the same if $|\lambda| = 1$, and decrease it if $|\lambda| < 1$. Moreover, the direction will be precisely the same if $\lambda > 0$, and flipped if $\lambda < 0$. If $\lambda = 0$, then the length of the arrow becomes zero.

### 4.4.4. Eigenvalues and eigenvectors of matrices: Characteristic Polynomial.

The eigenvalue equation for a matrix $A$ is

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

which is equivalent to

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

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

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

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

(c) Ángel Martin - August 2013
SOLVE-based Implementation. \{EIGEN\}

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

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

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

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

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

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

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

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

Below is the program listing for \{EIGEN\}. Note how the equation to solve already requires an auxiliary FAT entry, \{#EV\} – since a global label is always needed by FROOT. (You can refer to the SandMath manual if you need to refresh your understanding of FROOT and FINTG)
The program \texttt{EIGEN} works for N-dimensional orders. In that regard its execution time (provided that a decent initial guess is given) compares favorably to that of \texttt{CHRPO}, the other program that calculates eigenvalues. The difference of course is that \texttt{CHRPO} obtains all the eigen values simultaneously, whilst \texttt{EIGEN} does it one at a time.

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

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

When compared to other approaches, the program listed above is \textit{almost minimalistic} – that’s its real benefit and the reason that justifies its inclusion in the SandMatrix module. However relying on \texttt{FROOT} is clearly not a robust approach to calculate eigenvalues - The calculation of the characteristic polynomial using dedicated methods is a necessity.
\end{Verbatim}
3-Dimensional case. \( \{ \text{EV3x3}, \text{EV3} \} \)

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

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

\[
\det(A) - \text{tr}(A)\lambda + \lambda^2,
\]

For a \( 3 \times 3 \) matrix, the formula specifies the characteristic polynomial to be

\[
\det(A) - c_2\lambda + \text{tr}(A)\lambda^2 - \lambda^3.
\]

where \( c_2 \) is the sum of the principal minors of the matrix \( A \).

Given the above definitions it is clear now why functions \( \text{MDPS} \) and \( \Sigma_{IJJJ} \) will be helpful to obtain the coefficients of the characteristic polynomial for \( n=3 \). In effect, when using those functions the formulas change as follows: \( c_2 = (\text{MDPS} - \Sigma_{IJJJ}) \)

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

Example 1. Calculate the eigenvalues for \( A \), with \( a_{ij} = ij \)

Solution: \( pA(x) = 75,349 x^3 - 66 x^2 - 60 x = 0 \)

\[ x_1 = 66.890 \]
\[ x_2 = -0.897 \]
\[ x_3 = 2.24000E-9 \]

Example 2. Calculate the eigenvalues for \( A \), with \( a_{ij} = 1,2,3...9 \)

Solution: \( pA(x) = 0.076 x^3 - 15 x^2 - 18 x = 0 \)

\[ x_1 = 16.117 \]
\[ x_2 = -1.117 \]
\[ x_3 = 2.89100E-9 \]

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

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

The roots are obtained using the SandMath function \texttt{CROOT}, an all-MCODE implementation of the Cardano-Vieta formulas. Function \texttt{QROUT} is also used to display them.
General case: N-dimensional general matrix. \{ \texttt{CHRPOL} \}

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

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

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

The coefficients are calculated using the iterations:

\[
\begin{align*}
b_1 &= -\text{tr} (B_1), \quad \text{with } B_1 = \text{the original matrix, and} \\
b_k &= - \frac{\text{tr} (B_k)}{k}, \quad \text{with } B_k = A (B_{k-1} + b_{k-1} I), \quad k=2,\ldots n
\end{align*}
\]

The program works for orders \( n \) between 3 and 14. The case \( n=2 \) has a trivial solution \([\text{given by } b_2=1, b_1= \text{tr}(A), \text{and } b_0= -\text{det}(A)]\) ; therefore doesn't need to be included.

**Example.** Obtain the characteristic polynomial for the matrix \( A \) given below:

\[
A = \begin{bmatrix}
1 & -0.69 & 0.28 \\
-0.69 & 1 & 0.18 \\
0.28 & 0.18 & 1
\end{bmatrix}
\]

<table>
<thead>
<tr>
<th>Keystrokes</th>
<th>Display</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALPHA , &quot;AA&quot;, ALPHA</td>
<td>current X-reg</td>
<td>Matrix name in Alpha</td>
</tr>
<tr>
<td>3.003 , XEQ &quot;MATDIM&quot;</td>
<td>3.003</td>
<td>Creates matrix in X-Mem</td>
</tr>
<tr>
<td>XEQ “IMR”</td>
<td>“a1,1= ?”</td>
<td>Prompts for data, also showing current values</td>
</tr>
<tr>
<td>1, R/S</td>
<td>“a1,2= ?”</td>
<td></td>
</tr>
<tr>
<td>0.69, CHS, R/S</td>
<td>“a1,3= ?”</td>
<td></td>
</tr>
<tr>
<td>0.28, R/S</td>
<td>“a2,1”= ?</td>
<td></td>
</tr>
<tr>
<td>0.69, CHS, R/S</td>
<td>“a2,2= ?”</td>
<td></td>
</tr>
<tr>
<td>1, R/S</td>
<td>“a2,3= ?”</td>
<td></td>
</tr>
<tr>
<td>0.18, R/S</td>
<td>“a3,1= ?”</td>
<td></td>
</tr>
<tr>
<td>0.28, R/S</td>
<td>“a3,2= ?”</td>
<td></td>
</tr>
<tr>
<td>0.18, R/S</td>
<td>“a3,3= ?”</td>
<td>Last element</td>
</tr>
<tr>
<td>1, R/S</td>
<td>1.000</td>
<td></td>
</tr>
<tr>
<td>XEQ “CHRPOL”</td>
<td>“RUNNING...”</td>
<td>scrolls in the display, then</td>
</tr>
<tr>
<td>(* set F21</td>
<td>“x(aK*X^K)”</td>
<td>Reminder of convention</td>
</tr>
<tr>
<td>if you want AVIEW to stop each time</td>
<td>“a3=1,000000”</td>
<td>Coefficient of ( x^3 )</td>
</tr>
<tr>
<td>R/S</td>
<td>“a2=-3,000000”</td>
<td>Coefficient of ( x^2 )</td>
</tr>
<tr>
<td>R/S</td>
<td>“a1=2,413100”</td>
<td>Coefficient of ( x )</td>
</tr>
<tr>
<td>R/S</td>
<td>“a0=-0,343548”</td>
<td>First coef (independent term).</td>
</tr>
<tr>
<td>R/S</td>
<td>“X=0,180390390”</td>
<td>Scrolls in the display, then</td>
</tr>
<tr>
<td>R/S</td>
<td>“X=1,121568609”</td>
<td>First eigenvalue</td>
</tr>
<tr>
<td>R/S</td>
<td>“X=1,698238062”</td>
<td>Second eigenvalue</td>
</tr>
<tr>
<td>R/S</td>
<td>“X=1,698238062”</td>
<td>Third and last.</td>
</tr>
</tbody>
</table>
See the program code below in its entire splendor – realizing that it may be the last program written using Advantage Matrix functions...

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

```
1 LBL "CHRPOL" MNAME in Alpha 53 E3/E+ 1.001
2 DIM? n,00n 54 + 1.00(n+1) - cnt'l word
3 #? 55 "#"
4 -ADV MATRIX 56 PURFL
5 ASTO 01 MNAME 57 "p"
6 -CDD MATRIX shows 'RUNNING..." 58 PURFL
7 "|-,p" 59 VIEW for information
8 MAT# B = A 60 -CDD MATRIX shows 'RUNNING..."
9 ASWAP 61 PDEG new destination
10 DIM? n,00n 62 STO 00 as expected by RTSN
11 INT n 63 4
12 E 64 X<>Y? n<=n? 14 MDET independent term 66 CLX no, general case
13 + n+1 65 GTO 04 yes, particular case
14 MDET 66 GTO 00 go to EXIT
15 STO IND Y stored in Rn+1 67 E
16 ASWAP 68 + n+1
17 MAT# avoids LU issues 69 E6
18 DIM? 70 / 0,000100(n+1)
19 "#" auxiliary array 71 3 build the "from,to"
20 MATDIM 72 E3/E+ 1.003
21 FRC 0,00n 73 + 1.003100(n+1)
22 2 74 REGMOVE as expected by RTSN
23 + 2,00n 75 RTSN
24 STO 00 76 GTO 00 go to EXIT
25 CF 21 not halting VIEW 77 LBL 04
26 LBL 00 ←
27 VIEW 00 shows index 78 X<>Y? n<4?
28 "#" 80 RCL 02 a3
29 MIDN [#] = [I] 81 RCL 03 a2
30 "p" 82 RCL 04 a1
31 MTRACE tr (B) 83 RCL 05 a0
32 RCL 00 84 QUART
33 INT k+1 85 GTO 00 go to EXIT
34 E 86 LBL 03 ←
35 - k 87 RCL 01 a3
36 / 88 RCL 02 a2
37 CHS 89 RCL 03 a1
38 STO IND 00 pk = -tr (B) / k 90 RCL 04 a0
39 "X,#," 91 CROOT
40 MAT# [%] = pk [I] 92 "X="
41 "P,#,I" 93 ARCL Z
42 MAT+= [%] = [B] + p[I] 94 PROMPT real root
43 CLA 95 FC? 43 is RAD on?
44 ARCL 01 96 GTO 01 yes, complex roots
45 "|",#,# 97 X<> Z no, real roots
46 M*M B= A (B - pI) 98 CLX so we clear Z
47 ISG 00 99 X<> Z
48 GTO 00 100 LBL 01 ←
49 DIM? n,00n 101 QROUT output roots
50 FRC 0,00n 102 LBL 00 ←
51 E 103 MNAME? bring MNAME back
52 STO 01 it's monic (!) 104 END done
```

(c) Ángel Martin - August 2013
**Particular case: Symmetric Matrices** { **JACOBI** }

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

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

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

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

\[
\begin{align*}
    R_{pp} &= \cos z ; & R_{pq} &= \sin z ; & R_{qp} &= -\sin z ; & R_{qq} &= \cos z \\
    R_{ii} &= 1 ; & R_{ik} &= R_{iq} &= R_{ki} &= 0 ; & \text{for } i\neq p,q \text{ and } k\neq p,q
\end{align*}
\]

Let’s now denote: \( B = R^T A R \), which elements are as follows:

\[
\begin{align*}
    b_{ip} &= a_{ip} \cos z - a_{iq} \sin z \\
    b_{iq} &= a_{ip} \sin z + a_{iq} \cos z \\
    b_{ik} &= a_{ik} ; & \text{where } i,k \neq p,q
\end{align*}
\]

\[
\begin{align*}
    b_{pp} &= a_{pp} \cos^2 z + a_{qq} \sin^2 z - 2 a_{pq} \sin z \cos z \\
    b_{qq} &= a_{pp} \sin^2 z + a_{qq} \cos^2 z + 2 a_{pq} \sin z \cos z \\
    b_{pq} &= 0,
\end{align*}
\]

and the remaining elements are symmetric.

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

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

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

**Program remarks.** The accuracy and running times are display settings-dependent, however the computed eigenvalues are very often more accurate that it’d appear; for instance using FIX 5 it’s quite possible to have eigenvalues accurate to 8 decimal digits. The program is written to be as fast as possible and to occupy the minumim amount of program memory; the matrix is stored taking into account its symmetry, so that all elements are stored only once (as \( a_{ji} = a_{ij} \)). For a nxn matrix minimum size is \( \lceil \frac{1}{2} (n^2 + n) + 7 \rceil \).

**Example.** Find the eigenvalues for the 4x4 matrix: \( A = \)

\[
\begin{bmatrix}
    25 & -41 & 10 & -6 \\
    -41 & 68 & -17 & 10 \\
    10 & -17 & 5 & -3 \\
    -6 & 10 & -3 & 2
\end{bmatrix}
\]
### Example

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

<table>
<thead>
<tr>
<th>Keystrokes</th>
<th>Display</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>[\text{ALPHA} , &quot;AA&quot;, \text{ALPHA}]</td>
<td>current X-reg</td>
<td>Matrix name in Alpha</td>
</tr>
<tr>
<td>4.004, XEQ [\text{MATDIM}]</td>
<td>4.003</td>
<td>Creates mtrix in X-Mem</td>
</tr>
<tr>
<td>XEQ [\text{PMTM}]</td>
<td>(\text{R1: } -)</td>
<td>prompts for row-1</td>
</tr>
<tr>
<td>25, ENTER(^\wedge), CHS, 41, ENTER(^\wedge), (10), ENTER, CHS, 6, R/S</td>
<td>(\text{R2: } -)</td>
<td>prompts for row-2</td>
</tr>
<tr>
<td>CHS, 41, ENTER(^\wedge), 68, ENTER(^\wedge), CHS 17, ENTER(^\wedge), (10), R/S</td>
<td>(\text{R3: } -)</td>
<td>prompts for row-3</td>
</tr>
<tr>
<td>(10), ENTER(^\wedge), CHS, 17, ENTER(^\wedge), (5), ENTER(^\wedge), CHS, 3, R/S</td>
<td>(\text{R4: } -)</td>
<td>prompts for row-4</td>
</tr>
<tr>
<td>CHS, 6, ENTER(^\wedge), (10), ENTER(^\wedge), CHS, 3, ENTER(^\wedge), (2), R/S</td>
<td>XEQ [\text{CHRPOL}]</td>
<td>Scrolling on the display</td>
</tr>
<tr>
<td>(\text{RUNNING...})</td>
<td>(\Sigma(aK\times X^K))</td>
<td>Reminder of convention</td>
</tr>
<tr>
<td>(a4=1)</td>
<td>Coefficient of (x^4)</td>
<td></td>
</tr>
<tr>
<td>(a3=-100)</td>
<td>Coefficient of (x^3)</td>
<td></td>
</tr>
<tr>
<td>(a2=146)</td>
<td>Coefficient of (x^2)</td>
<td></td>
</tr>
<tr>
<td>(a1=-35)</td>
<td>Coefficient of (x)</td>
<td></td>
</tr>
<tr>
<td>(a0=1,00000)</td>
<td>First coef. (independent term)</td>
<td></td>
</tr>
<tr>
<td>(\text{RUNNING...})</td>
<td>Scrolling on the display</td>
<td></td>
</tr>
<tr>
<td>(X1=98,52170)</td>
<td>First root</td>
<td></td>
</tr>
<tr>
<td>(X2=1,18609)</td>
<td>Second root</td>
<td></td>
</tr>
<tr>
<td>(X3=0,25919)</td>
<td>Third root</td>
<td></td>
</tr>
<tr>
<td>(X4=0,03302)</td>
<td>Fourth and last root.</td>
<td></td>
</tr>
</tbody>
</table>

The solution is: \(\text{Chr}(A) = x^4 -100 \times x^3 + 146 \times x^2 - 35 \times +1\)
4.2.- Managing Polynomials.

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

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Input / Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>DTC</td>
<td>Deleting Tiny Coefficients</td>
</tr>
<tr>
<td>8</td>
<td>“P+P”</td>
<td>Polynomial Sum</td>
</tr>
<tr>
<td>9</td>
<td>“P-P”</td>
<td>Polynomial Subtraction</td>
</tr>
<tr>
<td>10</td>
<td>“P^nP”</td>
<td>Product of Polynomials</td>
</tr>
<tr>
<td>11</td>
<td>“P/P”</td>
<td>Division of Polynomials</td>
</tr>
<tr>
<td>12</td>
<td>PCPY</td>
<td>Polynomial Copy</td>
</tr>
<tr>
<td>13</td>
<td>PDIV</td>
<td>Euclidean Division</td>
</tr>
<tr>
<td>14</td>
<td>PEDIT</td>
<td>Edits Polynomial Coefficients</td>
</tr>
<tr>
<td>15</td>
<td>PMTP</td>
<td>Prompts for Coefs in Alpha List</td>
</tr>
<tr>
<td>16</td>
<td>PPRD</td>
<td>Polynomial Multiplication</td>
</tr>
<tr>
<td>17</td>
<td>PSUM</td>
<td>Polynomial Addition &amp; Subtraction</td>
</tr>
<tr>
<td>18</td>
<td>PVAL</td>
<td>Polynomial Evaluation</td>
</tr>
<tr>
<td>19</td>
<td>PVIEW</td>
<td>Views Polynomial Coefficients</td>
</tr>
</tbody>
</table>

4.2.1. Defining and Storing Polynomials.

A polynomial is an expression of the form

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

where \( a(n) \neq 0 \)

Or, more concisely:

\[ \sum_{i=0}^{n} a_i x^i \]

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

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

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

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

A nice feature is that it’ll check for available data registers to complete all the storage, re-adjusting the calculator SIZE if necessary. **PEDIT** does not use any data registers itself.
Note that PEDIT includes in the prompts the current value held in the corresponding data register, so you don't need to type a new one if it's already correct. Alternatively you can use PVIEW to review the coefficients without any editing capabilities. In this mode the prompts don't have the question mark at the end, which indicates the value cannot be changed from the program.

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

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

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

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

```
1 LBL "PEDIT"
2 SF 08 flags mode
3 ENTER^ copies cntl word to Y
4 I<>J swaps bbb and eee
5 E
6 +
7 SIZE? current size
8 X->Y not enough?
9 X>Y? not enough?
10 PSIZE adjust size
11 RDN
12 RDN cntl word to X-reg
13 GTO 00 skip over
14 LBL "PVIEW"
15 CF 00 flags mode
16 LBL 00
17 -ADV POLYN shows convention
18 PSE
19 ENTER^ copies cntl word to Y
20 PDEG polyn degree
21 X->Y cntl word to X-reg
22 STO L saves it in L
23 X->Y degree to X-reg
24 LBL 01
25 "a"
26 AIP append index
```

(c) Ángel Martin - August 2013
4.2.2. Polynomial Arithmetic \{ $\text{PSUM}$, $\text{PPRD}$, $\text{PDIV}$ \}

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

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

<table>
<thead>
<tr>
<th>Operation</th>
<th>Addtion, Subtraction, Multiplication</th>
<th>Euclidean Division</th>
<th>Copy</th>
</tr>
</thead>
</table>
| **Input** | bbb.eee1 in Z  
bb.eee2 in Y  
1st. Reg of result in X | bbb.eee of dividend in Y  
bb.eee of divisor in Y | bbb.eee of source in Y  
bb or destination in X |
| **Output** | bbb.eee of result in X  
bb.eee of reminder in Y  
bb.eee of quotient in X | bbb.eee or result in X |

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

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

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

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

$a(x) = 2.x^5 + 5.x^4 - 21.x^3 + 23.x^2 + 3.x + 5$  
and $b(x) = 2.x^2 - 3.x + 1$

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

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

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

$a(x) = 2.x^5 + 5.x^4 - 21.x^3 + 23.x^2 + 3.x + 5$  
and $b(x) = 2.x^2 - 3.x + 1$

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

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

The solutions are: $q(x) = x^3 + 4.x^2 - 5.x + 2$  
and $r(x) = 14.x + 3$
Example 3. - Calculate the addition and subtraction of the polynomials \( a(x) \) and \( b(x) \) below:

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

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

The solutions are:

\[
a(x) + b(x) = 4x^3 + x^2 + 12x + 7 \\
a(x) - b(x) = 7x^2 - 2x + 5
\]

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

```
1  LBL "P+P"  32  LBL 10
2  CF 01  33  "N#1?" order P1
3  GTO 00  34  PROMPT n1
4  LBL "P/P"  35  4
5  SF 01  36  +
6  LBL 00  37  E3/E+ 1,00(n+4)
7  XEQ 10 combined data entry  38  3
8  FC 01 product?  39  +  4,00(n+4)
9  GTO 00 yes, go there  40  STO 00
10  RND division  41  PEDIT
11  PDIV  42  XEQ 05 adjust index
12  X<>Y reminder ctrl word  43  ENTER^ push stack
13  STO 00 store  44  "N#2?" order P2
14  X<>Y  45  PROMPT n2
15  PVIEW show quotient  46  + n2+eee1
16  X<> 00  47  I<>J 0,00(n2+eee1)
17  GTO 02  48  + (eee1+1),00(eee1+n2)
18  LBL 00 multiplication  49  PEDIT
19  PPRD  50  RCL 00 bbb.eee1
20  GTO 02  51  X<>Y bbb.eee2
21  LBL "P-P"  52  LBL 05
22  CF 01  53  ENTER^ bbb.eee2
23  GTO 01  54  I<>J eee.bbb2
24  LBL "P-P"  55  INT eee2
25  SF 01  56  E
26  LBL 01  57  + eee2+1
27  XEQ 10 combined data entry  58  END
28  PSUM  59
29  LBL 02  
30  PVIEW show result (reminder)
31  RTN done
```
4.2.3. Deleting tiny Coefficients. \{DTC\}

Evaluating and Copying Polynomials. \{PVAL, PCPY\}

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

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

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

When using \texttt{PCPY} be careful that the register ranges for both polynomials do not overlap.
4.3. Polynomial Root Finders.

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

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

### 4.3.1. Quartic Equation solutions. { QUART }

**QUART** solves the equation \( x^4 + a \cdot x^3 + b \cdot x^2 + c \cdot x + d = 0 \)

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

The method used can be summarized as follows:

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

\[
x^4 + p \cdot x^2 + q \cdot x + r = 0 \quad (E')
\]

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

\[
x = \pm \sqrt{2} z_1^{1/2} \cdot \text{sign}(-q) \pm \sqrt{2} ( z_2^{1/2} + z_3^{1/2} )
\]

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

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

**Example1:** Solve \( x^4 - 2 \cdot x^3 - 35 \cdot x^2 + 36 \cdot x + 180 = 0 \)

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

\[ X1 = 6,000, \quad X2 = 3,000 \]

\[ X3 = -2,000, \quad X4 = -5,000 \]
Example2: Solve \( x^4 - 5x^3 + 11x^2 - 189x + 522 = 0 \)
-5 ENTER, 11 ENTER, -189 ENTER, 522 , XEQ “QUART” >>>>
\( Z = -2 + j5,000 \) (note how true integer values don't display zeros after the decimal point)
\( X3 = 3,000, \quad X4 = 3,000 \)

Example3: Solve \( x^4 - 8x^3 + 26x^2 - 168x + 1305 = 0 \)
-8 ENTER, 26 ENTER, -168 ENTER, 1305 , XEQ “QUART” >>>>
\( Z = -2 + j5,000 \) (note how true integer values don't display zeros after the decimal point)
\( Z = 6 + j3,000 \)
4.3.2. General case: degree N. (PRoot, RTSN, BAIRS)

Given a polynomial $P$,

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

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

$$P(z) = P''(z) Q(z) + R(z); \text{ with } P''(z) = [\sum bi z^{n-i}] , i=2,1...(n-2)$$

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

Let $Q(x) = x^2 + px + q$; and $R(x) = rx + s$

Then the reduced polynomial coefficients are given by

$$b_i = a(i-2) - p b(i-1) - q b(i-2) ; \quad i = 2, 3, ..., (n+2) \quad (1)$$

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

$$r = b(n+1)$$
$$s = b(n+2) + p b(n+1) \quad (2)$$

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

The problem is thus obtaining the coefficients $p,q$ of such a quotient polynomial that would cancel the reminder (i.e. that make $r=0$ and $s=0$. This is accomplished by using an iterative approach, starting with some initial guesses for them ($p_0, q_0$), iterating until there is no change in two consecutive values,

$$r'(p,q) + r = 0; \quad \text{or: } r'(p,q) = -r$$
$$s'(p,q) + s = 0; \quad \text{or: } s'(p,q) = -s$$

Expressing it using their partial derivarives it results:

$$dp \left( \frac{\delta r}{\delta p} \right) + dq \left( \frac{\delta r}{\delta q} \right) = -r$$
$$dq \left( \frac{\delta s}{\delta p} \right) + dq \left( \frac{\delta s}{\delta q} \right) = -s$$

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

From equation (1) above it follows:

$$\frac{\delta b_i}{\delta p} = ci = -b(i-1) - p c(i-1) - q c(i-2); \quad i = 2,3..., (n+2)$$
$$\frac{\delta b_i}{\delta q} = c(i-1)$$

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

$$c(n+1) \frac{dp}{cn} + [c(n+1) + p c n] \frac{dq}{dn} = -[b(n+2) + p b(n+1)] \quad (3)$$
Starting with \((p_0=0.5; q_0=0.5)\) as initial guesses we'll obtain \(dp\) and \(dq\) for each pair of values \((p,q)\).

With them we adjust the previous guess, so that the new corrected values for \(p\) and \(q\) are

\[
p' = p + dp
q' = q + dq
\]

This will be repeated until the precision factor \(\varepsilon\) is smaller than the convergence criteria; The precision factor is calculated as follows:

\[
\varepsilon = \frac{\text{abs}(dp) + \text{abs}(dq)}{\text{abs}(p) + \text{abs}(q)}
\]

The program dimensions and populates matrices \([RS]\) and \([CN]\) to hold the current values of \(p,q\) and the coefficients \(C_n\) respectively:

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

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

**Program Details.**

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

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

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

**Example 1.** Find the five roots of the polynomial below

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

<table>
<thead>
<tr>
<th>Keystrokes</th>
<th>Display</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>XEQ “PROOT”</td>
<td>“ORDER=?”</td>
<td>Prompts for the degree</td>
</tr>
<tr>
<td>5, R/S</td>
<td>“(\Sigma(aK*X^K))”</td>
<td>Reminder of convention</td>
</tr>
<tr>
<td>2, R/S</td>
<td>“a5= ?”</td>
<td>prompts for coeffs, showing current</td>
</tr>
<tr>
<td>7, R/S</td>
<td>“a4= ?”</td>
<td></td>
</tr>
<tr>
<td>20, R/S</td>
<td>“a3= ?”</td>
<td></td>
</tr>
<tr>
<td>81, R/S</td>
<td>“a2= ?”</td>
<td></td>
</tr>
<tr>
<td>190, R/S</td>
<td>“a1= ?”</td>
<td></td>
</tr>
<tr>
<td>150, R/S</td>
<td>“a0= ?”</td>
<td></td>
</tr>
<tr>
<td>“Y”</td>
<td>“STO? Y/N”</td>
<td>prompts for storage option</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

At this point the different precision factors are shown, which should be decreasing as the iterations converge towards the solutions – and this repeated as many times at quadratic factors are needed.
The solutions are shown below (in FIX 5):

\[ Z = -2,00000 + J1,00000 \text{ and its conjugate (not shown)} \]
\[ Z = 1,00000 + J3,00000 \text{ and its conjugate (not shown)} \]
\[ Z = -1,50000 \]

And the matrix \([\text{ROOTS}]\) is left in X-Mem, with 5 rows and two columns, as follows:

\[
\begin{bmatrix}
-2 & 1 \\
-2 & -1 \\
1 & 3 \\
1 & -3 \\
-1.5 & 0 \\
\end{bmatrix}
\]

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

Bairstow Method.

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

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

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

If \(\deg(p)\) is odd, we have
\[ p(x) = (a_{0}.x+b).(x^2+u_1.x+v_1)......(x^2+u_m.x+v_m) \]
with \(m = (n-1)/2\)

If \(\deg(p)\) is even
\[ p(x) = (a_0x^2+u_1.x+v_1)(x^2+u_2.x+v_2)......(x^2+u_m.x+v_m) \]
with \(m = n/2\)

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

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

Using \([\text{PMTP}]\) to store the coefficients beginning in R09, thus the control word is \(9,015\)

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

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

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

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

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

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

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

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

The original listing was originally published in the AECROM manual, and it's reproduced here practically unaltered.
4.4.2. Polynomial interpolation. \{ POLINT \}

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

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

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

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

- You start the program and enter x at the prompt. \texttt{XEQ "POLINT"}
- Then you enter the first data pair, preferably one which has an x_i close to x. \texttt{x0, R/S}
  \texttt{y0, R/S}
  The program returns y0.

- You enter another data pair. \texttt{R/S}
  The program returns the results of a linear interpolation. \texttt{x1, R/S}
  \texttt{y1, R/S}

- You enter another data pair. \texttt{R/S}
  The program returns the results of a quadratic interpolation. \texttt{x2, R/S}
  \texttt{y2, R/S}

- You enter another data pair. \texttt{R/S}
  The program returns the results of a cubic interpolation. \texttt{x3, R/S}
  \texttt{y3, R/S}

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

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

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

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

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

<table>
<thead>
<tr>
<th>T/ K</th>
<th>380</th>
<th>400</th>
<th>450</th>
<th>480</th>
<th>500</th>
<th>530</th>
<th>560</th>
</tr>
</thead>
<tbody>
<tr>
<td>p/ MPa</td>
<td>0.12885</td>
<td>0.24577</td>
<td>0.93220</td>
<td>1.7905</td>
<td>2.6392</td>
<td>4.4569</td>
<td>7.1062</td>
</tr>
</tbody>
</table>

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

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

The program listing is shown below.

```
1  LBL "POLINT"  33  X<>Y
2  FC? 55  34  AIP
3  SF 21  35  X<>Y
4  "X=?”  36  "Y=?”
5  PROMPT  37  PROMPT
          x value of point  prompts for Yk
6  STO 00  38  DSE 02
7  3.05  39  GTO 02
8  STO 01  40  LBL 03
9  LBL 01  41  RCL IND 02
10  RCL 01  42  *
11  INT  43  LASTX
12  k  44  RCL Z
13  -  45  -
14  E3/E+  46  ISG 02
15  1.00(k-1)  47  RCL IND 02
16  +  48  LASTX
17  4.00(k-1)  49  *
18  STO 02  50  ST- Z
19  RCL 01  51  LASTX
20  INT  52  RDN
21  k  53  RDN
22  -  54  /
23  k-3  55  LBL 02
24  /  56  STO IND 01
25  AIP  57  ISG 02
26  "Y=?”  58  GTO 03
27  PROMPT  59  "Y=?”
          prompts for Xk
28  RCL 00  60  ARCL X
29  -  61  AVIEW
30  STO IND 01  62  ISG 01
31  ISG 01  63  GTO 01 next order
32  "Y=?”  64  END done
```

(c) Ángel Martin - August 2013
4.4.3. Prime Factors Decomposition \{ PRMF , PF>X , TOTNT \}

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

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Input / Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>PRMF</td>
<td>Argument in X-reg</td>
</tr>
<tr>
<td>2</td>
<td>PF&gt;X</td>
<td>Prime factors in Matrix file</td>
</tr>
<tr>
<td>3</td>
<td>TOTNT</td>
<td>Argument in X-reg</td>
</tr>
</tbody>
</table>

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

You can use \texttt{PF>X} to check the result; it re-builds the original argument from the values in the \texttt{PRMF} matrix, using the obvious relationship:

\[ X = \prod_{i=1}^{\text{primes}} \text{PF}(i)^{m(i)} \]

\textbf{Euler's Totient function.}

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

\[ \begin{array}{|c|c|c|}
\hline
n & \text{PF} & \phi(n) \\
\hline
1,477 & 7*211 & 1,260 \\
819,735 & 3*5*7*37*211 & 362,880 \\
123,456 & 2^6*3*643 & 41,088,000 \\
\hline
\end{array} \]

\textbf{Examples.} Calculate the prime factors and the totient for the following numbers:
The programs are listed below. There's no fancy algorithm for \([\text{TOTNT}]\), it just counts the number of prime factors after doing the decomposition as a preliminary step.

<table>
<thead>
<tr>
<th>1</th>
<th>LBL &quot;TOTNT&quot;</th>
<th>Euler's Totient Function</th>
<th>55</th>
<th>GTO 03</th>
<th>skip if yes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>SF 04</td>
<td>flag case</td>
<td>56</td>
<td>ST/ L</td>
<td>divide number by PF</td>
</tr>
<tr>
<td>3</td>
<td>XEQ 10</td>
<td>get all Prime Factors</td>
<td>57</td>
<td>LASTX</td>
<td>Reduced number</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td></td>
<td>58</td>
<td>GTO 00</td>
<td>loop back</td>
</tr>
<tr>
<td>5</td>
<td>MSU</td>
<td>sets pointer to 1:1</td>
<td>59</td>
<td>LBL 03</td>
<td>Store Exponent</td>
</tr>
<tr>
<td>6</td>
<td>X&lt;&gt;Y</td>
<td>argument to x</td>
<td>60</td>
<td>RCL 00</td>
<td>recover PF</td>
</tr>
<tr>
<td>7</td>
<td>LBL 07</td>
<td></td>
<td>61</td>
<td>MSR+</td>
<td>store in matrix</td>
</tr>
<tr>
<td>8</td>
<td>MRC+</td>
<td>get element</td>
<td>62</td>
<td>GTO 01</td>
<td>next factor</td>
</tr>
<tr>
<td>9</td>
<td>1/X</td>
<td>invert it</td>
<td>63</td>
<td>LBL 02</td>
<td>New PF found</td>
</tr>
<tr>
<td>10</td>
<td>CHS</td>
<td>sign change</td>
<td>64</td>
<td>STO 01</td>
<td>Store for comparisons</td>
</tr>
<tr>
<td>11</td>
<td>E</td>
<td></td>
<td>65</td>
<td>RCL 00</td>
<td>previous exponent</td>
</tr>
<tr>
<td>12</td>
<td>+</td>
<td>add 1</td>
<td>66</td>
<td>MSR+</td>
<td>Store Old PF Exponent</td>
</tr>
<tr>
<td>13</td>
<td>*</td>
<td>multiply</td>
<td>67</td>
<td>RDN</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>FC? 09</td>
<td>end of row?</td>
<td>68</td>
<td>ST/ L</td>
<td>divide number by PF</td>
</tr>
<tr>
<td>15</td>
<td>GTO 07</td>
<td>loop back</td>
<td>69</td>
<td>LASTX</td>
<td>Reduced number</td>
</tr>
<tr>
<td>16</td>
<td>CLD</td>
<td>refresh display</td>
<td>70</td>
<td>DIM?</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>RTN</td>
<td>done.</td>
<td>71</td>
<td>X&lt;&gt;Y</td>
<td>Bring the new PF back</td>
</tr>
<tr>
<td>18</td>
<td>LBL &quot;PRMF&quot;</td>
<td>Prime Factors</td>
<td>72</td>
<td>MSR+</td>
<td>store new PF</td>
</tr>
<tr>
<td>19</td>
<td>CF 04</td>
<td>flag case</td>
<td>73</td>
<td>FSTC 00</td>
<td>Was it Prime?</td>
</tr>
<tr>
<td>20</td>
<td>LBL 10</td>
<td></td>
<td>74</td>
<td>GTO 01</td>
<td>Bail Out, we're done</td>
</tr>
<tr>
<td>21</td>
<td>&quot;PRMF&quot;</td>
<td></td>
<td>75</td>
<td>X&lt;&gt;Y</td>
<td>Bring the number back</td>
</tr>
<tr>
<td>22</td>
<td>&quot;+&quot;</td>
<td></td>
<td>76</td>
<td>GTO 05</td>
<td>Start Over</td>
</tr>
<tr>
<td>23</td>
<td>E3/E+</td>
<td>1,002</td>
<td>77</td>
<td>LBL &quot;PF&gt;X&quot;</td>
<td>Rebuild number</td>
</tr>
<tr>
<td>24</td>
<td>MATDIM</td>
<td>Create Matrix</td>
<td>78</td>
<td>SF 04</td>
<td>flag case</td>
</tr>
<tr>
<td>25</td>
<td>CLX</td>
<td></td>
<td>79</td>
<td>&quot;PRMF&quot;</td>
<td>matrix name</td>
</tr>
<tr>
<td>26</td>
<td>MSUA</td>
<td>sets pointer to 1:1</td>
<td>80</td>
<td>SF 10</td>
<td>fake condition</td>
</tr>
<tr>
<td>27</td>
<td>RDN</td>
<td>argument to x</td>
<td>81</td>
<td>LBL 01</td>
<td>PF Completed</td>
</tr>
<tr>
<td>28</td>
<td>CF 00</td>
<td>default: not prime</td>
<td>82</td>
<td>E</td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>INT</td>
<td>condition x</td>
<td>83</td>
<td>FCT 10</td>
<td>end of matrix?</td>
</tr>
<tr>
<td>30</td>
<td>ABS</td>
<td>to avoid errors</td>
<td>84</td>
<td>MSR+</td>
<td>store it as last exp.</td>
</tr>
<tr>
<td>31</td>
<td>PRIME?</td>
<td>is it prime?</td>
<td>85</td>
<td>STO 01</td>
<td>initial value</td>
</tr>
<tr>
<td>32</td>
<td>SF 00</td>
<td>FIRST PF found</td>
<td>86</td>
<td>MSUA</td>
<td>sets pointer to 1:1</td>
</tr>
<tr>
<td>33</td>
<td>MSR+</td>
<td>Store this PF</td>
<td>87</td>
<td>CLA</td>
<td>Clean Slate</td>
</tr>
<tr>
<td>34</td>
<td>X=0?</td>
<td>is PF =1?</td>
<td>88</td>
<td>LBL 06</td>
<td>Rebuild the number</td>
</tr>
<tr>
<td>35</td>
<td>GTO 01</td>
<td>yes, leave the boat</td>
<td>89</td>
<td>MRR+</td>
<td>get prime factor</td>
</tr>
<tr>
<td>36</td>
<td>FSTC 00</td>
<td>Was it Prime?</td>
<td>90</td>
<td>FC 04</td>
<td>if not totient case</td>
</tr>
<tr>
<td>37</td>
<td>GTO 01</td>
<td>if Prime, we're done</td>
<td>91</td>
<td>AIP</td>
<td>add it to Alpha</td>
</tr>
<tr>
<td>38</td>
<td>STO 01</td>
<td>Store PF for comparisons</td>
<td>92</td>
<td>MRR+</td>
<td>get multiplicity</td>
</tr>
<tr>
<td>39</td>
<td>ST/ L</td>
<td>divide number by PF</td>
<td>93</td>
<td>FC 04</td>
<td>if not totient and/</td>
</tr>
<tr>
<td>40</td>
<td>LASTX</td>
<td>Reduced number</td>
<td>94</td>
<td>X=1?</td>
<td>or if it is one</td>
</tr>
<tr>
<td>41</td>
<td>LBL 05</td>
<td></td>
<td>95</td>
<td>GTO 04</td>
<td>skip adding to Alpha</td>
</tr>
<tr>
<td>42</td>
<td>E</td>
<td>reset counter</td>
<td>96</td>
<td>&quot;[^-^]&quot;</td>
<td>otherwise put symbol</td>
</tr>
<tr>
<td>43</td>
<td>STO 00</td>
<td></td>
<td>97</td>
<td>AIP</td>
<td>and add it to the string</td>
</tr>
<tr>
<td>44</td>
<td>RDN</td>
<td></td>
<td>98</td>
<td>LBL 04</td>
<td></td>
</tr>
<tr>
<td>45</td>
<td>LBL 00</td>
<td></td>
<td>99</td>
<td>Y^X</td>
<td>PF^Exp</td>
</tr>
<tr>
<td>46</td>
<td>RCL 01</td>
<td>recall PF</td>
<td>100</td>
<td>ST* 00</td>
<td>Rebuilding the number</td>
</tr>
<tr>
<td>47</td>
<td>X&lt;&gt;Y</td>
<td>Reduced number</td>
<td>101</td>
<td>FS710</td>
<td>End of Array?</td>
</tr>
<tr>
<td>48</td>
<td>PRIME?</td>
<td>is it prime?</td>
<td>102</td>
<td>GTO 04</td>
<td>yes, leave the boat</td>
</tr>
<tr>
<td>49</td>
<td>SF 00</td>
<td>PF found</td>
<td>103</td>
<td>FC 04</td>
<td>if not totient case</td>
</tr>
<tr>
<td>50</td>
<td>XHY?</td>
<td>Compare this and old PF's</td>
<td>104</td>
<td>&quot;[^-^]&quot;</td>
<td>append symbol</td>
</tr>
<tr>
<td>51</td>
<td>GTO 02</td>
<td>skip over if different</td>
<td>105</td>
<td>GTO 06</td>
<td>next PF</td>
</tr>
<tr>
<td>52</td>
<td>ISG 00</td>
<td>Same One</td>
<td>106</td>
<td>LBL 04</td>
<td></td>
</tr>
<tr>
<td>53</td>
<td>NOP</td>
<td>Increase counter</td>
<td>107</td>
<td>RCL 00</td>
<td>final result</td>
</tr>
<tr>
<td>54</td>
<td>FSTC 00</td>
<td>Was it Prime?</td>
<td>108</td>
<td>FC 04</td>
<td>if not totient case</td>
</tr>
<tr>
<td>55</td>
<td>AVIEW</td>
<td></td>
<td>109</td>
<td>Show the construct</td>
<td></td>
</tr>
<tr>
<td>56</td>
<td>END</td>
<td></td>
<td>110</td>
<td></td>
<td>done.</td>
</tr>
</tbody>
</table>
4.4.4. Polynomial Fitting {POLFIT}

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

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

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

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

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

\[ y = P(x) = a_1 + a_2 x + a_3 x^2 + a_4 x^3 + \ldots + a_{n+1} x^n \]

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

Program listing

```
01  LBL "POLFIT" to use, simply XEQ "POLFIT"
02  "N=?” prompts for the degree N of the polynomial
03  PROMPT .. and waits for the user to enter N
04  1 add 1 to get the number of data points
05  + N+1
06  1.001 the required multiplier
07  * forms the matrix dimensions [N+1].00[N+1]
08  "MX" specifies matrix MX to be created in X-Mem
09  MATDIM creates and dimensions matrix MX in X-MEM
10  0 specifies first row, first column and ..
11  MSIJ .. resets the row/column indexes
12  LBL 00 loop to ask for data & compute MX elements
13  MRIJ .. recalls the current value of the indexes
14  "X” forms the prompt to ask the user to enter xi
15  AIP .. appends the index to the prompt
16  "/=?” appends “=” to the prompt
17  PROMPT prompts to enter xi and resume execution
18  ENTER^ fills the stack with the value of xi ..
19  ENTER^ in order to compute all powers of xi ..
20  ENTER^ from 1 to xi^n and store them in MX
21  1 initializes the value of xi^0 [i.e.: 1]
22  MSR+ stores it in MX and updates the indexes
```
23  LBL 01    loop to compute the powers of xi 24  * computes $x_i^j$
25  MSR+    stores it in MX and updates the indexes
26  FC? 09    are we done with this row?
27  GTO 01    not yet, go back for the next xi power
28  FC? 10    row done. Are we done with all rows?
29  GTO 00    not yet, go back to ask for the next xi
30  CLA    all rows done, MX complete. Make it current
31  DIM?    get its dimensions: \([N+1].00\)\([N+1]\]
32  INT    get N+1 (avoid using a register)
33  "MY"    specify vector MY to be created in X-MEM
34  MATDIM    creates and dimensions vector MY in X-MEM
35  LBL B    ask for yi data and store them in MY
36  0    specifies 1st element of the vector and ... 37  MSIJ    resets the index to the 1st element
38  LBL 02    loop for next data and store them in MY
39  MRIJ    recalls the current value of the index
40  "Y"    forms the prompt to ask for yi
41  AI P ..    appends the index to the prompt
42  "/=?"    appends "="?" to the prompt
43  PROMPT    prompts the user to enter yi
44  MSR+    stores it in MY and updates the index
45  FC? 10    are we done with all elements?
46  GTO 02    not yet, go back to ask for the next yi
47  "MX,MY"    all yi stored. Specify MX,MY for the system
48  MSYS    solves the system for the coefficients
49  LBL C    retrieve and display each coeff.
50  0    specifies 1st element of the coeffs. vector
51  MSIJ    resets the index to the 1st coefficient
52  LBL 03    loop to retrieve the next coefficient
53  MRIJ    recalls the current value of the index
54  "A"    forms the prompt to display each coeff.
55  AI P ..    appends the index to the prompt
56  "/="    appends "=" to the prompt
57  MRR+    retrieves the value of the current coeff.
58  ARCL X    appends the value to the prompt
59  PROMPT    shows the value to the user
60  FC? 10    are we done outputting all the coeffs?
61  GTO 03    not yet, go back for the next coefficient
62  END    all done. End of execution.

Notes

- As the Advantage ROM can work with matrices directly in X-Mem, [POLFIT] doesn't use any main RAM registers and so it will run even at SIZE 000. This has the added advantage (pun intended) of avoiding any register conflicts with other programs.

- POLFIT creates two matrices in X-Mem, namely [MX] and [MY], which aren't destroyed upon termination. Retaining [MX] allows the user to compute the coefficients of another polynomial using the same x data but different y data. In that case, the x data need not be entered again, only the new y data must be entered. Further, as the MX matrix is left in LU-decomposed form after the first fit, the second fit will proceed much faster. Retaining [MY] allows the user to employ the polynomial for interpolating purposes, root finding, numeric integration or differentiation, etc.

- Lines 2-11 prompt the user for the degree of the polynomial, then allocate the system matrix in Extended Memory ([MATDIM]) and reset the indexes ([MSIJ]).
• Lines 12-22 set up a loop that will fill up the rows of \([\text{MX}]\). Notice the use of the miscellaneous function AIP to build the prompt, and MSR+ to store the value and automatically advance the indexes to point to the next element.

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

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

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

**Example**

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

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

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

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

\[
P(x) = \cos(5 \arccos x) = 5 x - 20 x^3 + 16 x^5
\]

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

\text{CAT"4} \quad \text{"MX M036" [the system matrix is 6x6 = 36 elements]}
\text{"MY M006" [the coeff. matrix is 6x1 = 6 elements]}
554.0000 \quad \text{[this value varies with your configuration]}
4.4.5. Orthogonal Polynomial Fit. \{OPFIT\}

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

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

The method followed uses the construct \(Y(x) = d_0 P_0(x) + d_1 P_1(x) + \ldots + d_n P_n(x)\); where \(p_0, p_1, \ldots, p_n\) are the orthogonal polynomials corresponding to the entered data that satisfy the expression \(\sum p_i P_j = 0, \text{ for every } i \neq j\)

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

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

\[ P(x) = x^4 - 2x^3 + 3x^2 - 4x + 5 \]

<table>
<thead>
<tr>
<th>(x_i)</th>
<th>-3</th>
<th>-2</th>
<th>-1</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>5</th>
<th>6</th>
<th>7560</th>
</tr>
</thead>
<tbody>
<tr>
<td>(y_i)</td>
<td>179</td>
<td>57</td>
<td>15</td>
<td>5</td>
<td>3</td>
<td>9</td>
<td>47</td>
<td>165</td>
<td>435</td>
<td>953</td>
<td>1839</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

The results are all provided in the table below:

<table>
<thead>
<tr>
<th>Degree ((n))</th>
<th>Corrlt. ((r^2))</th>
<th>Errors ((e^2))</th>
<th>Determin. ((d^2))</th>
<th>Coefficients</th>
</tr>
</thead>
<tbody>
<tr>
<td>(n = 1)</td>
<td>R1=4,482218E-1</td>
<td>E0=3,295160E5 E1=1,818179E5</td>
<td>D0=3,370000E2 D1=1,228000E2</td>
<td>a0=9,140000E1 a1=1,228000E2</td>
</tr>
<tr>
<td>(n = 2)</td>
<td>R2=9,000134E-1</td>
<td>E2=3,294720E4</td>
<td>D2=4,000000E1</td>
<td>a0=-1,486000E2 a1=-3,720000E1 a2=4,000000E1</td>
</tr>
<tr>
<td>(n = 3)</td>
<td>R3=9,821452E-1</td>
<td>E3=5,883429E3</td>
<td>D3=6,000000E0</td>
<td>a0=1,700000E1 a1=-7,200000E1 a2=4,000000E0 a3=6,000000E0</td>
</tr>
<tr>
<td>(n = 4)</td>
<td>R4=1,000000E0</td>
<td>E4=0,000000E0</td>
<td>D4=1,000000E0</td>
<td>a0=5,000000E0 a1=-4,000000E0 a2=3,000000E0 a3=-2,000000E0 a4=1,000000E0</td>
</tr>
</tbody>
</table>

Original author: [OPFIT] was written by Eugenio Úbeda, and published in the UPLE. The version in the SandMatrix has only minimal changes made to it. It is by far the longest program in the module.
4.4.6. From Poles to Zeros... and back. \{ POLZER, PFE \}

These two programs complete the applications section. The first one calculates the zeros of a polynomial expressed as a partial expansion of factors, as would typically be the case when working with transfer functions in control theory. The second program builds the partial fraction expansion for a polynomial given its “standard” (or natural) form.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Input / Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>POLZER</td>
<td>Zeros of transfer functions</td>
</tr>
<tr>
<td>2</td>
<td>PFE</td>
<td>Partial Fraction Expansion</td>
</tr>
</tbody>
</table>

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

\[ P(x) = \sum \left(\frac{1}{x-p_i}\right) ; \quad i = 1,2,\ldots n \]

Which will be transformed into:

\[ P(x) = \sum a_i x^i ; \quad i = 0,1,\ldots (n-1) \]

The coefficients are obtained using the following formulae:

\[
\begin{align*}
a(n-1) &= n \\
a(n-2) &= (n-1) \sum p_i \\
a(n-3) &= (n-2) \sum \sum p_i p_j \\
a(n-4) &= (n-3) \sum \sum \sum p_i p_j p_k \\
a(n-5) &= (n-4) \sum \sum \sum \sum p_i p_j p_k p_l \\
a(n-6) &= (n-5) \sum \sum \sum \sum \sum p_i p_j p_k p_l p_m 
\end{align*}
\]

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

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

\[ G(s) = \frac{1}{s} + \frac{1}{s-1} + \frac{1}{s-2} + \frac{1}{s-3} + \frac{1}{s-4} \]

**Keystrokes**

```
XEQ "POLZER"  #POL=?
5, R/S  P(1)=?
0, R/S  P(2)=?
1, R/S  P(3)=?
2, R/S  P(4)=?
3, R/S  P(5)=?
4. R/S  "Σ ... ΣΣΣ ... ΣΣΣΣΣ... ΣΣΣΣΣΣ... "
"CFS? Y/N"
```

```
"Y"
```

```
a(4)=5,000000
R/S  a(3)=-40,000000
R/S  a(2)=105,000000
R/S  a(1)=-100,000000
R/S  a(0)=24,000000
```

Therefore the “natural” polynomial form is as follows:

\[ G(s) = 5 s^4 - 40 s^3 + 105 s^2 - 100 s + 24 \]
Next the execution is transferred to **RTSN**, which will calculate the roots following the iterative process explained in section 4.3.1. Remember that the accuracy is dictated by the number of decimals places set.

<table>
<thead>
<tr>
<th>R/S</th>
<th>“RUNNING...”</th>
</tr>
</thead>
<tbody>
<tr>
<td>R/S</td>
<td>Z=0.35557</td>
</tr>
<tr>
<td>R/S</td>
<td>Z=1.45609</td>
</tr>
<tr>
<td>R/S</td>
<td>Z=2.54395</td>
</tr>
<tr>
<td>R/S</td>
<td>Z=3.64442</td>
</tr>
</tbody>
</table>

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

### 4.4.7. Partial Fraction Decomposition

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

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

\[
\frac{f(x)}{g(x)} \quad \Rightarrow \quad \sum_{j} \frac{f_j(x)}{g_j(x)}
\]

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

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

### Implementation

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

Given a rational function \( R(x) = \frac{P(x)}{Q(x)} \) with \( Q(x) = \prod_{i} q_i(x) \) and \( \gcd(q_i, q_j) = 1 \quad \text{for all} \quad i \neq j \), this program returns the partial fraction expansion:

\[
R(x) = E(x) + \frac{p_{1,1}(x)}{q_1(x)} \cdot x^{i_1} + \frac{p_{1,2}(x)}{q_1(x)} \cdot x^{i_1-1} + \cdots + \frac{p_{1,\mu_1}(x)}{q_1(x)}
\]

\[
+ \cdots + \frac{p_{n,1}(x)}{q_n(x)} \cdot x^{i_n} + \frac{p_{n,2}(x)}{q_n(x)} \cdot x^{i_n-1} + \cdots + \frac{p_{n,\mu_n}(x)}{q_n(x)}
\]
where \( \deg p_i < \deg q_i \), and \( E(x) \) is the quotient in the Euclidean division \( P(x) = E(x) Q(x) + p(x) \) and \( p(x) \) is the remainder.

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

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

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

**Example 1.** Calculate the partial fraction decomposition for \( R(x) \) below.

\[
R(x) = \frac{P(x)}{Q(x)} = \frac{(6x^5 - 19x^4 + 20x^3 - 7x^2 + 7x + 10)}{[(2x^2 + x + 1)(x - 2)^2]}
\]

<table>
<thead>
<tr>
<th>Keystrokes</th>
<th>Display</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>XEQ &quot;PFE&quot;</td>
<td>&quot;#DEN=?”</td>
<td>Input number of factors</td>
</tr>
<tr>
<td>2, R/S</td>
<td>&quot;NUM#=?”</td>
<td>inputs degree of numerator</td>
</tr>
<tr>
<td>5, R/S</td>
<td>&quot;( \Sigma(aK*X^K))&quot;</td>
<td>Reminder of convention</td>
</tr>
<tr>
<td>6, R/S</td>
<td>&quot;a5= ?&quot;</td>
<td>coefficients data entry</td>
</tr>
<tr>
<td>19, CHS, R/S</td>
<td>&quot;a3= ?&quot;</td>
<td></td>
</tr>
<tr>
<td>20, R/S</td>
<td>&quot;a2= ?&quot;</td>
<td></td>
</tr>
<tr>
<td>7, CHS, R/S</td>
<td>&quot;a1=?”</td>
<td></td>
</tr>
<tr>
<td>7, R/S</td>
<td>&quot;a0=?&quot;</td>
<td></td>
</tr>
<tr>
<td>10, R/S</td>
<td>&quot;Q1#=?”</td>
<td>Input degree of Q1 in den.</td>
</tr>
<tr>
<td>2, R/S</td>
<td>&quot;( \Sigma(aK*X^K))&quot;</td>
<td>Reminder of convention</td>
</tr>
<tr>
<td>2, R/S</td>
<td>&quot;a2=?&quot;</td>
<td></td>
</tr>
<tr>
<td>1, R/S</td>
<td>&quot;a1=?&quot;</td>
<td></td>
</tr>
<tr>
<td>1, R/S</td>
<td>&quot;a0=?&quot;</td>
<td></td>
</tr>
<tr>
<td>1, R/S</td>
<td>&quot;Q2#=?&quot;</td>
<td></td>
</tr>
<tr>
<td>1, R/S</td>
<td>&quot;( \Sigma(aK*X^K))&quot;</td>
<td>Reminder of convention</td>
</tr>
<tr>
<td>1, R/S</td>
<td>&quot;a1=?&quot;</td>
<td></td>
</tr>
<tr>
<td>1, R/S</td>
<td>&quot;a0=?&quot;</td>
<td></td>
</tr>
<tr>
<td>2, R/S</td>
<td>&quot;XP^\mu&quot;</td>
<td>time to enter the multiplicities now</td>
</tr>
<tr>
<td>1, R/S</td>
<td>&quot;a0=?&quot;</td>
<td>exponent of first factor</td>
</tr>
<tr>
<td>2, R/S</td>
<td>&quot;a1= ?”</td>
<td>exponent of second factor</td>
</tr>
<tr>
<td>R/S</td>
<td>&quot;a0=1”</td>
<td>end of data output.</td>
</tr>
</tbody>
</table>

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

1. The cnt’l word stored in R15 is for the Quotient polynomial, \( E(x) \)

2. The cnt’l word in R05 gives the entire register range for the coefficients of all the \( p(x) \) polynomials – the numerators of the expanded fractions. It needs to be interpreted depending on the denominators \( q(x) \) are polynomials of degree 1 or polynomials pf degree 2 with negative discriminant. The contents of these registers are to be read
by groups of 1 number if \( \deg q_j = 1 \) the numerators are constants
by groups of 2 numbers if \( \deg q_j = 2 \) the numerators are polynomials of degree 1
by groups of 3 numbers if \( \deg q_j = 3 \) the numerators are polynomials of degree 2, and so on ...

3. The third in \( R_0 \) is for an alternative solution using a new reminder \( p(x) \)

Thus in this case registers \( R_{16} \) and \( R_{17} \) contain the coefficients for \[ E(x) = 3x + 1 \]
And registers \( R_{33} - R_{36} \) for the denominator polynomials: (which must be three of them!)

\[
p_{1,1}(x) = 2x + 3 \quad p_{2,1}(x) = 4 \quad p_{2,2}(x) = 5
\]

Thus the final result is as follows:

\[
R(x) = \frac{E(x)}{2x^2 + x + 1} + \frac{p_{1,1}(x)}{(x-2)^2} + \frac{p_{2,2}(x)}{x-2}
\]

Or alternatively using the data in registers \( R_{18} - R_{21} \) (cnt’l word in \( Z \)):

\[
p(x) = 12x^3 - 12x^2 - 5x + 6
\]

and therefore:

\[
R(x) = \frac{E(x)}{Q(x)} + \frac{p(x)}{Q(x)}
\]

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

\[
R(x) = \frac{P(x)}{Q(x)} = \frac{x^5}{(3x^2 + 1)^2}
\]

The three control words returned are:

\[
Z: 18.021 \quad \text{with:} \quad R_{18} = -2/3, \quad R_{19} = 0, \quad R_{10} = -1/9, \quad R_{21} = 0
\]
\[
Y: 28.031 \quad \text{with} \quad R_{29} = 1/9, \quad R_{29} = 0, \quad R_{30} = -2/9, \quad R_{31} = 0
\]
\[
X: 16.017 \quad \text{with:} \quad R_{16} = 1/9 \quad \text{and} \quad R_{17} = 0
\]

The range in \( Y \) must be split as \( p_{1,2} = x/9 \quad x + 0 \); and \( p_{2,2} = -2x/9 + 0 \)

Therefore:

\[
R(x) = \frac{E(x)}{(3x^2 + 1)^2} + \frac{p_{1,2}(x)}{(3x^2 + 1)} + \frac{p_{2,2}(x)}{(3x^2 + 1)}
\]

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

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

[http://www.hp41.org/LibView.cfm?Command=View&ItemID=776](http://www.hp41.org/LibView.cfm?Command=View&ItemID=776)
There are a few new M-Code functions in the SandMatrix that make direct usage of the module’s subroutines. A representative example is given below, showing the very short routine LU? – that checks whether the matrix is in its decomposed form – simply by reading the appropriate digit in the matrix header register.

1. LU? Header A5FA 0BF "LU" Jumps to Bank_2
2. LU? Header A5FB 015 "U" adds "4" to [XS]
3. LU? Header A5FC 00C "L" [LNCHD]
4. LU? Header A5FD 379 PORT DEP: Jumps to Bank_2
5. LU? A5FE 03C XQ [LNCHD]
6. LU? A5FF 1D9 ->A5D9 [LSK]
7. LU? A600 388 <parameter> [LSK]
8. LU? A601 00B JNC +01 [LSK]
9. LU? A602 100 ENROM1 restore bank-1
10. LU? A603 080 C=N ALL header register
11. LU? A604 25C PT= 9 LU digit
12. LU? A605 2E2 ?CRD @PT
13. LU? A606 089 ?NC GO False
14. LU? A607 05A ->162E [LSK]
15. LU? A608 065 ?NC GO True
16. LU? A609 05A ->1619 [LSK]

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

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

1. ^MROW Header B658 097 "W" Input Matrix Row
2. ^MROW Header B659 00F "O" Ángel Martin
3. ^MROW Header B65A 012 "R" ^MROW B65D 0C4 CLRF 10 start anew: no CHS yet
4. ^MROW Header B65B 00D "M" start anew: no commas yet
5. ^MROW Header B65C 01E "^" start anew: no digits yet
6. ^MROW ^MROW B65D 0C4 CLRF 10 start anew: no CHS yet
7. ^MROW B65E 184 CLRF 11 start anew: no commas yet
8. ^MROW B65F 344 CLRF 12 start anew: no digits yet
9. ^MROW B660 0F8 READ 3(X)
10. ^MROW B661 070 N=C ALL Clears Alpha
11. ^MROW B662 345 ?NC XQ [CLA]
12. ^MROW B663 040 ->10D1 Build Msg - all cases
14. ^MROW B665 0FC ->3F85 [APRMSG2]
15. ^MROW B666 212 "R" [APRMSG2]
16. ^MROW B667 080 C=N ALL row number in BCD format
17. ^MROW B668 37C RCR 12 move the MSB to C(0)
18. ^MROW B669 21C PT= 2
19. ^MROW B66A 010 LD@PT- 0
20. ^MROW B66B 2D0 LD@PT- B add colon to digit
21. ^MROW B66C 3E8 WRIT 15(e) write it in display (9-bit)
22. ^MROW B66D 355 ?NC XQ blank space to LCD
23. ^MROW B66E 03C ->0FDS DSPALL
24. ^MROW B66F 33D ?NC GO [AUST]
25. ^MROW B670 112 ->44CF [AUST]

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

(c) Ángel Martin - August 2013
ALIST BCKARW 44CD 055 ?NC KO<br>  Delete char plus logic
ALIST 44CE 116 ->4515 [DELCAR]
ALIST 44CF 115 ?NC KQ<br>  Partial Data Entry!
ALIST 44D0 038 ->0E45 [NEXT1]
ALIST 44D1 3E3 JNC -04 [BCKARW]
ALIST 44D2 00C ?FSET 3 numeric input?
ALIST 44D3 093 JNC +18d NO, KEEP LOOKING
ALIST 44D4 08E A<>C MS recall LS digit from A[13]
ALIST 44D5 130 LDI S&BX <@
ALIST 44D6 003 CON: pre-load the numeric mask
ALIST 44D7 3E8 WRIT 15(e) write it in display (9-bit)
ALIST 44D8 348 SETF 12 enable SPACE
ALIST 44D9 024 ->0952 [ENCP00]
ALIST 44DA 39C PT= 0
ALIST 44DB 058 G=C @PT,+/ reset PTEMP bits
ALIST 44DC 149 ?NC XQ Disable PER, enable RAM [ENCPO0]
ALIST 44DD 01C ->07F6 [ENLCD]
ALIST ANCHOR1 44DE 35B JNC ONE PROMPT
ALIST 44DF 28C ?FSET 7 decimal key pressed?
ALIST 44EA 03B JNC +07d NO, KEEP LOOKING
ALIST 44EB 3D9 ?NC XQ Enable Display (not cleared)
ALIST 44EC 01C ->07F6 [ENLCD]
ALIST ANCHOR2 44ED 35B JNC -21d add to Alpha
ALIST 44EE 130 LDI S&BX <@
ALIST 44EF 106 A<>C S&X [TOALPH]
ALIST 44F0 030 CON: ENTER^ keycode [030]
ALIST 44F1 366 ?ARC S&BX (1A420)
ALIST 44F2 04F JC +09 add proper radix sign
ALIST 44F3 34C ?FSET 12 digits input already?
ALIST ANCHOR1 44F4 383 JNC -16d ONE PROMPT
ALIST 44F5 04C CLRF 10 clear CHS flag
ALIST 44F6 184 CLRF 11 allow RADIX
ALIST 44F7 344 CLRF 12 set SPACE flag
ALIST 44F8 355 ?NC XQ add space to LCD [DSPL20]
ALIST 44F9 02C ->0FDS [APNDNW]
ALIST 44FA 393 JNC -14d add to Alpha
ALIST 44FB 130 LDI S&BX <@
ALIST 44FC 02D CON: R/S keycode [370]
ALIST 44FD 366 ?ARC S&BX terminate digit entry
ALIST 44FE 07B JNC +15d [WAYOUT]
ALIST 44FF 130 LDI S&BX <@
ALIST 4500 230 CON: CHS keycode [230]
ALIST 4501 366 ?ARC S&BX (1A420)
ALIST 4502 023 JNC +04 [BLINK1]
ALIST 4503 265 ?NC XQ Blink Display - pass #2
ALIST 4504 020 ->0899 [BLINK1]
ALIST 4505 37B JNC -17d ONE PROMPT
ALIST 4506 0CC ?FSET 10 been used already?
ALIST 4507 3F7 JC -02 ONE PROMPT
ALIST 4508 0C8 SETF 10 first time
ALIST 4509 130 LDI S&BX <@
ALIST 450A 02D ... append "="
ALIST 450B 3E8 WRIT 15(e) 9-bit LCD write
ALIST 450C 303 JNC -32d [TOALPH]
ALIST WAYOUT 450D 32D ?NC XQ LEFT-justify LCD [LEFTJ]
ALIST 450E 0AC ->2BF7 [LEFTJ]
### SandMatrix_4 Manual

<table>
<thead>
<tr>
<th>ALIST</th>
<th>ALIST</th>
<th>ALIST</th>
<th>ALIST</th>
</tr>
</thead>
<tbody>
<tr>
<td>450F</td>
<td>461</td>
<td>7NC XQ</td>
<td>Clear LCD and reset things</td>
</tr>
<tr>
<td>4510</td>
<td>124</td>
<td>-4958</td>
<td>[EXIT3]</td>
</tr>
<tr>
<td>4511</td>
<td>175</td>
<td>7NC XQ</td>
<td>Adjust F10 Status</td>
</tr>
<tr>
<td>4512</td>
<td>171</td>
<td>&gt;E5D0</td>
<td>[RADF10]</td>
</tr>
<tr>
<td>4513</td>
<td>31D</td>
<td>7NC GO</td>
<td>Normal Function ReturnKB</td>
</tr>
<tr>
<td>4514</td>
<td>002</td>
<td>&gt;00C7</td>
<td>[NEPKB]</td>
</tr>
<tr>
<td>4515</td>
<td>3BB</td>
<td>READ 14(d)</td>
<td>to delete rightmost chr</td>
</tr>
<tr>
<td>4516</td>
<td>158</td>
<td>M=C ALL</td>
<td>save it for later</td>
</tr>
<tr>
<td>4517</td>
<td>149</td>
<td>7NC XQ</td>
<td>Disable PER, enable RAM</td>
</tr>
<tr>
<td>4518</td>
<td>024</td>
<td>&gt;X052</td>
<td>[ENCPOD]</td>
</tr>
<tr>
<td>4519</td>
<td>178</td>
<td>READ S(M)</td>
<td></td>
</tr>
<tr>
<td>451A</td>
<td>2E</td>
<td>?C80 ALL</td>
<td>anything in Alpha?</td>
</tr>
<tr>
<td>451B</td>
<td>037</td>
<td>JC +06</td>
<td>yes, go on</td>
</tr>
<tr>
<td>451C</td>
<td>104</td>
<td>CLRF 8</td>
<td>no, abort if empty</td>
</tr>
<tr>
<td>451D</td>
<td>104</td>
<td>CLRF 8</td>
<td></td>
</tr>
<tr>
<td>451E</td>
<td>070</td>
<td>&gt;16C2</td>
<td>Mainframe Message</td>
</tr>
<tr>
<td>451F</td>
<td>03C</td>
<td>&quot;NULL&quot;</td>
<td>from table</td>
</tr>
<tr>
<td>4520</td>
<td>37B</td>
<td>JNC -17d</td>
<td>Reset everything and leave</td>
</tr>
<tr>
<td>4521</td>
<td>2E5</td>
<td>7NC XQ</td>
<td>remove last Alpha char</td>
</tr>
<tr>
<td>4522</td>
<td>010</td>
<td>&gt;A000</td>
<td>[ALPHA]</td>
</tr>
<tr>
<td>4523</td>
<td>198</td>
<td>CiM ALL</td>
<td>Recall deleted char value</td>
</tr>
<tr>
<td>4524</td>
<td>106</td>
<td>A=C S&amp;X</td>
<td>store in A for comparisons</td>
</tr>
<tr>
<td>4525</td>
<td>130</td>
<td>LD1 S&amp;X</td>
<td>check for SPACE</td>
</tr>
<tr>
<td>4526</td>
<td>020</td>
<td>&quot;space&quot;</td>
<td>&lt;space&gt;</td>
</tr>
<tr>
<td>4527</td>
<td>04D</td>
<td>7NC XQ</td>
<td>Complete the logic</td>
</tr>
<tr>
<td>4528</td>
<td>114</td>
<td>&gt;4F2</td>
<td>[CHUNKE]</td>
</tr>
<tr>
<td>4529</td>
<td>881</td>
<td>7NC GO</td>
<td>repeat the prompt</td>
</tr>
<tr>
<td>452A</td>
<td>112</td>
<td>&gt;-44E0</td>
<td>[G0DACK]</td>
</tr>
<tr>
<td>452B</td>
<td>866</td>
<td>?AFC S&amp;X</td>
<td>carry if different</td>
</tr>
<tr>
<td>452C</td>
<td>01F</td>
<td>JC + 03</td>
<td></td>
</tr>
<tr>
<td>452D</td>
<td>548</td>
<td>SETF 12</td>
<td>allow new space entry</td>
</tr>
<tr>
<td>452E</td>
<td>043</td>
<td>JNC +20d</td>
<td>BAIL OUT</td>
</tr>
<tr>
<td>452F</td>
<td>130</td>
<td>LD1 S&amp;X</td>
<td>check for &quot;.&quot; chr</td>
</tr>
<tr>
<td>4530</td>
<td>02D</td>
<td>&quot;,&quot;</td>
<td>&quot;,&quot; chr value</td>
</tr>
<tr>
<td>4531</td>
<td>366</td>
<td>?AFC S&amp;X</td>
<td>carry if not &quot;,&quot;</td>
</tr>
<tr>
<td>4532</td>
<td>02F</td>
<td>JC + 05</td>
<td></td>
</tr>
<tr>
<td>4533</td>
<td>04C</td>
<td>IFSET 12</td>
<td>is there SPACE chr?</td>
</tr>
<tr>
<td>4534</td>
<td>017</td>
<td>JC +02</td>
<td></td>
</tr>
<tr>
<td>4535</td>
<td>0C4</td>
<td>CLRF 10</td>
<td>allow new &quot;.&quot; entry</td>
</tr>
<tr>
<td>4536</td>
<td>063</td>
<td>JNC +12d</td>
<td>BAIL OUT</td>
</tr>
<tr>
<td>4537</td>
<td>198</td>
<td>CiM ALL</td>
<td>Recall deleted char value</td>
</tr>
<tr>
<td>4538</td>
<td>3D8</td>
<td>C=SS ST XP</td>
<td>Got a radix? if so, we need to</td>
</tr>
<tr>
<td>4539</td>
<td>14C</td>
<td>?IFSET 6</td>
<td>replace it without comma</td>
</tr>
<tr>
<td>453A</td>
<td>043</td>
<td>JNC +0B</td>
<td></td>
</tr>
<tr>
<td>453B</td>
<td>3D9</td>
<td>7NC XQ</td>
<td>Enable Display (not cleared)</td>
</tr>
<tr>
<td>453C</td>
<td>01C</td>
<td>&gt;07FS</td>
<td></td>
</tr>
<tr>
<td>453D</td>
<td>044</td>
<td>CLRF 6</td>
<td>Remove the radix value</td>
</tr>
<tr>
<td>453E</td>
<td>284</td>
<td>CLRF 7</td>
<td>(both if need be)</td>
</tr>
<tr>
<td>453F</td>
<td>3D8</td>
<td>C=SS ST XP</td>
<td>recall deleted char value</td>
</tr>
<tr>
<td>4540</td>
<td>3EB</td>
<td>WRIT 15(e)</td>
<td>write it in display</td>
</tr>
<tr>
<td>4541</td>
<td>184</td>
<td>CLRF 11</td>
<td>Re-allow comma writing</td>
</tr>
<tr>
<td>4542</td>
<td>3E0</td>
<td>RTN</td>
<td></td>
</tr>
<tr>
<td>4543</td>
<td>549</td>
<td>7NC XQ</td>
<td>Disable PER, enable RAM</td>
</tr>
<tr>
<td>4544</td>
<td>024</td>
<td>&gt;0952</td>
<td>[ENCPOD]</td>
</tr>
<tr>
<td>4545</td>
<td>3BB</td>
<td>READ 14(d)</td>
<td>put F28 to F9</td>
</tr>
<tr>
<td>4546</td>
<td>2BC</td>
<td>RCR 7</td>
<td></td>
</tr>
<tr>
<td>4547</td>
<td>24B</td>
<td>SET 9</td>
<td></td>
</tr>
<tr>
<td>4548</td>
<td>11E</td>
<td>C=CC ALL</td>
<td>comma or period?</td>
</tr>
<tr>
<td>4549</td>
<td>013</td>
<td>JNC +02</td>
<td>overflows if COMMA (cf28)</td>
</tr>
<tr>
<td>454A</td>
<td>244</td>
<td>CLRF 9</td>
<td>comma = CF 2B</td>
</tr>
<tr>
<td>454B</td>
<td>3D9</td>
<td>7NC XQ</td>
<td>Enable Display (not cleared)</td>
</tr>
<tr>
<td>454C</td>
<td>01C</td>
<td>&gt;076S</td>
<td></td>
</tr>
<tr>
<td>454D</td>
<td>3BB</td>
<td>READ 14(d)</td>
<td>read right</td>
</tr>
<tr>
<td>454E</td>
<td>3D8</td>
<td>C=SS ST XP</td>
<td></td>
</tr>
<tr>
<td>454F</td>
<td>148</td>
<td>SET 6</td>
<td></td>
</tr>
<tr>
<td>4550</td>
<td>24C</td>
<td>IFSET 9</td>
<td>comma or period?</td>
</tr>
<tr>
<td>4551</td>
<td>013</td>
<td>JNC +02</td>
<td></td>
</tr>
<tr>
<td>4552</td>
<td>28B</td>
<td>SET 7</td>
<td>should replace the last chr</td>
</tr>
<tr>
<td>4553</td>
<td>3D8</td>
<td>C=SS ST XP</td>
<td>with the same one w/ radix</td>
</tr>
<tr>
<td>4554</td>
<td>3EB</td>
<td>WRIT 15(e)</td>
<td>9-bit LCD write</td>
</tr>
<tr>
<td>4555</td>
<td>130</td>
<td>LDI S&amp;X</td>
<td></td>
</tr>
<tr>
<td>4556</td>
<td>02C</td>
<td>&quot;&quot;</td>
<td>appends &quot;,&quot; [02C]</td>
</tr>
<tr>
<td>4557</td>
<td>24C</td>
<td>IFSET 9</td>
<td></td>
</tr>
<tr>
<td>4558</td>
<td>360</td>
<td>7C RTN</td>
<td>no need, return</td>
</tr>
<tr>
<td>4559</td>
<td>026</td>
<td>C=1 S&amp;X</td>
<td></td>
</tr>
<tr>
<td>455A</td>
<td>26C</td>
<td>C=1 S&amp;X</td>
<td>appends &quot;,&quot; [02C]</td>
</tr>
<tr>
<td>455B</td>
<td>360</td>
<td>RTN</td>
<td></td>
</tr>
</tbody>
</table>

(c) Angel Martin - August 2013
The End.

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

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

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