Math Pac 2 – My Comments and Proposals

The ***HP-71 Math Pac 2*** is a new, *undergoing*, exciting project of **Jean-François Garnier**[1], the renowned author of ***Emu71/DOS***, a fantastic *HP-71B* emulator which I've used extensively over the years to create many of my articles, challenges and programming projects, even to quickly and effortlessly develop a preliminary model for math-oriented algorithms included in a big professional project. And I got explicit recognition from the client for it !

The ***HP-71 Math Pac 2*** is a dream come true for the ones who, like myself, love the *HP-71B* and its amazing original *Math Pac*, *aka* the *Math ROM*. The combination, when used within the excellent and eminently usable *Emu71/DOS*, provides an environment which is ideal for both quick interactive calculations as well as quickly modeling math-oriented applications.

Even in the modern world of powerful software running on fast processors, there are still times when you can get results faster and with far less fuss using *Emu71/DOS* on any device instead of turning on a computer, waiting for the OS to load and all the miriad daily updates being applied, starting *MS Visual Studio* (say), creating a new project, writing the code, debugging it, compiling it, then running it. And then creating an installation package if you want to use the executable somewhere else.

Now I've been very carefully considering what would be the most useful additions to the original *Math Pac* set which would still fit in the 5 Kb or so of unused *ROM* space while optimizing the usefulness and minimizing the implementation difficulty and this article is the set of **comments and proposals** I came up with, possibly followed by a sequel article *implementing* them as optimized *BASIC* code to serve as a most useful reference for eventual conversion to *Assembler* (in the form of new keywords and/or *binary* subprograms), while in the meantime they can be called from user programs in their *BASIC* form.

This document is structured in the following sections:

1. **New Additions Proposed**

   The *Array Statements (14)*, *Scalar-Valued Array Functions (3)* and *Real Scalar Functions (5)* which I propose for inclusion in the ***Math Pac2***. Each has full syntax, one or more examples, and a rationale for its inclusion.

2. **Extensions to Existing Keywords**

   The extended functionality I propose for some already existing keywords *(2)* to enhance their functionality.

3. **Additional Aliases**

   The aliases I propose for some already existing keywords *(2)* to enhance their ease of invocation.

4. **Low-priority Additional Functions**

   These are the functions *(5)* already included in the current ***Math Pac 2*** which I propose be given very low priority or better still, to be removed altogether for the reasons given.

5. **Conclusion**

   My thanks to **Jean-François** for committing to this project and my best wishes for its successful completion.

[1] Visit **Jean-François**' *"The Math ROM for the HP-71B"* web page `(www.jeffcalc.hp41.eu/emu71/mathrom)` for the extensive details on both the original *Math Pac* and his new *Enhanced Math LEX*, *aka* the ***"Math Pac 2"*** project.

1. New Additions Proposed

The following 22 additional keywords are proposed for their inclusion in the ***Math Pac 2***. Most (but not all) will be described just for *real* arguments and results for simplicity but the the vast majority of them can/should be coded to handle *complex* values as well with little extra work.

These proposals are the ones I consider the most generally useful, *carefully selected* after I checked the 100+ programs I've written for the *HP-71* over the last 36 years on all subjects, from math to games to professional projects. I identified the parts I had to write as *BASIC* code which would greatly benefit from being assembly-language keywords instead, and which aren't covered in existing *ROM*s (such as the *JPC ROM*, for instance). Everyone will surely have their own picks but these are mine and I offer a *rationale* for their inclusion.

Keyword List				
Keyword	***Pag.***	***Description*** **Array Statements**	***Difficulty***	***Priority***
MAT..SORT	3	Sorts an array in increasing or decreasing order, various options	Medium	Higher
MAT..SHUF	4	Shuffles an array into random order (inverse of MAT..SORT)	Low	Normal
MAT..REV	5	Reverses the order of all or part of the array elements	Very Low	Normal
MAT..REPT	6	Extracts the real parts of the array elements	Very Low	Lower
MAT..IMPT	6	Extracts the imaginary parts of the array elements	Very Low	Higher
MAT..CONJ	7	Returns the conjugates of the array elements	Very Low	Normal
MAT..ABS	8	Returns the absolute values of the array elements	Very Low	Higher
MAT..RAN	9	Fills up an array with random values, either real or integers in a range	Very Low	Higher
MAT..ROUND	10	Returns the array elements rounded to N places	Very Low	Higher
MAT..^	11	Raises a square matrix to an integer power	Very Low	Normal
MAT..PCHAR	12	Returns the Characteristic Polynomial of a square matrix *(eigenvalues)*	Medium	Higher
MAT..PDER	13	Returns the coefficients of a polynomial's N^{th} derivative	Low	Normal
MAT..PFIT	14	Returns the coefficients of the Exact Polynomial Fit (Collocation)	Medium	Higher
MAT..PFITLS	15	Returns the coefficients of the Least Squares Polynomial Fit	Medium	Higher
Scalar-Valued Array Functions				
PEVAL	16	Evaluates a polynomial for a given argument	Very Low	Higher
DINTG	17	Computes the integral of a discrete set of datapoints	Low	Normal
TRACE	18	Returns the *Trace* of a square matrix	Very Low	Lower
Real Scalar Functions				
LAMW	19	Evaluates the *Lambert W* function for a given argument and branch	Medium	Higher
AGM	20	Evaluates the *Arithmetic-Geometric Mean* of two arguments	Low	Higher
ROUND	21	Returns its scalar argument rounded to N places	Very Low	Higher
RNDG	22	Returns a random number subject to a Gaussian distribution	Low	Normal
MODP	23	Computes the Modular Exponentiation even for large exponents	Medium	Higher

Notes:

- Difficulty : Very Low can be done in **1** line of *BASIC* code, Low in **2** lines, Medium in **3-6** lines.
- Priority : Higher most useful & slower/less accurate if in *BASIC*, Lower less used, doable in *BASIC*.

Estimated size: all 22 keywords should take about ***3-3.5 Kb*** of *ROM* space or less to implement.

SORT **Matrix Sort**

```
MAT A=SORT(B [,D [,C [,U,V]])
```

Where **A** and **B** are arrays and **D**, **C**, **U** and **V** are scalar variables.

Sorts the elements of array **B** in ascending (default, **D** = 0) or descending (**D** ≠ 0) order, using column **C** as the sorting column (default is the first column) and sorting only from row **U** to row **V** (default is from first to last row), and places the result in array **A**.

Syntax examples:

`MAT A=SORT(B)`	Sorts all rows of B in ascending order using the 1st column elements.
`MAT A=SORT(B,1)`	Sorts all rows of B in descending order using the 1st column's elements.
`MAT A=SORT(B,1,3)`	Sorts all rows of B in descending order using the 3rd column's elements.
`MAT A=SORT(B,1,3,10,20)`	Sorts rows 10 to 20 of B in descending order using the 3rd column's elements.

Not usable in CALC mode.

Example

Sort in place and in ascending order the first 3 rows of this matrix by the elements of its 3rd column:

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

```
DESTROY ALL @ OPTION BASE 1 @ STD [END LINE]
DIM A(5,4) @ MAT INPUT A [END LINE]
A(1,1) ?
          3,1,4,1,5,9,2,6,5,3,5,8,9,7,9,3,2,3,8,4 [END LINE]

MAT A=SORT(A,0,3,1,3) [END LINE]
MAT DISP A;
```

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

Rationale: Sorting data is among the most frequent operations and doing it using *BASIC* code is a very slow, looping affair which is a real bother to code. For large datasets one would strive for relatively complex, quick algorithms such as *Quicksort* or *Heapsort*, but much more frequently when using the *HP-71* one needs to sort datasets having just a few elements, frequently under 100 elements or so, in which case the most efficient sorting algorithms aren't really needed and something like ***Insert Sort*** is perfectly adequate. Even some variant of *Shell Sort* won't be too complex and can be used proficiently as well.

Implementing this keyword using *Insert Sort* (or *Shell Sort*) will take the drudge out of sorting and will be much faster than coding the algorithms in *BASIC*. And for under 100 elements, more than fast enough.

SHUF **Matrix Shuffle**

MAT **A**=SHUF(**B**)

Where **A** and **B** are arrays.

Assigns to **A** the array **B** after being randomly shuffled. **A** and **B** are usually the same array.

Not usable in CALC mode.

Examples

1) To be used in a card game, fill up a vector with the integers 1 to 52 representing the deck of cards, randomly shuffle it (using **1** as the seed for the *RNG*) and deal three hands of 5 cards each from the shuffled deck.

```
DESTROY ALL @ OPTION BASE 1 @ RANDOMIZE 1 @ STD END LINE
DIM D(52) @ FOR I=1 TO 52 @ D(I)=I @ NEXT I @ MAT D=SHUF(D) END LINE

FOR H=1 TO 3 @ FOR I=1 TO 5 @ DISP D(I); @ NEXT I @ DISP @ NEXT H END LINE
```

```
 17  12  29  43   5    first hand
 15   6  35  26  49    second hand
  9  18  32  10   2    third hand
```

2) Some sky survey is to be conducted at angles 0º, 10º, ..., 180º, but to avoid any possible systematic bias the survey won't proceed in the natural increasing angle order but with the angles shuffled at random. Display the new angle order to be used instead:

```
DESTROY ALL @ OPTION BASE 1 @ RANDOMIZE 1 @ STD END LINE
DIM A(19) @ FOR I=1 TO 19 @ A(I)=10*(I-1) @ NEXT I

MAT A=SHUF(A) @ FOR I=1 TO 19 @ DISP A(I); NEXT I @ DISP END LINE
```

```
 130  60  170  0  30  120  90  40  150  20  140  180  10  100  80  160  110  50  70
```

Rationale: *Shuffling* is the functional inverse of *Sorting* and there are many areas of all types, including both recreational and scientific/engineering, where it's necessary to conduct a stochastic simulation in which items are to be randomly selected to be acted upon but the selection must be *without replacement*, i.e.: all items must be eventually selected at random and there must be no duplicate selections. Real-life applications include games (card games, lotteries, bingo, dominoes, etc.), probability, simulation of financial markets, avoidance of unknown systematic bias, selection of samples for biology experiments, etc.

The **MAT..SHUF** keyword does the random shuffle at assembler speeds and, which is more, not only does it avoid having to use a slow *BASIC* loop but also avoids the naïve technique for shuffling which most people tend to use when not knowing better, i.e.: doing N exchanges of elements at random, which usually results in either underestimating or overestimating the number of exchanges needed, thus ruining the randomness or wasting time or both, apart from the fact that this method *does* introduce biases. The implementation of **MAT..SHUF** needs minimal resources and avoids all those pitfalls.

REV **Vector Reversion**

```
MAT A=REV(B [,U [,V]])
```

Where **A** and **B** are vectors and **U**, **V** are scalar variables.

Reverses the order of all the elements of vector **B** and places them in the corresponding locations of vector **A** (usually **A** and **B** will be the same vector so reversion will take place *in situ*).

If the optional parameter **U** is supplied, the elements from index **U** to the end of the vector will be reversed. If additionally the optional parameter **V** is supplied, just the elements from index **U** to index **V** will be reversed. Examples:

`MAT A=REV(A)`	Reverses in place the order of all the elements of **A**.
`MAT A=REV(A,5)`	Reverses in place the order of the elements of **A** starting at index 5.
`MAT A=REV(A,5,9)`	Reverses in place the order of the elements of **A** from index 5 to index 9.

Not usable in CALC mode.

Examples

1) Reverse in place the order of the 2nd and 3rd elements of the following column vector:

A = [3 5 9 2] (it's a *column* vector but printed horizontally to save space)

```
DESTROY ALL @ OPTION BASE 1 @ STD END LINE
DIM A(4) @ MAT INPUT A END LINE
A(1) ?
       3,5,9,2 END LINE
MAT A=REV(A,2,3) @ MAT DISP A; END LINE
```

A = [3 9 5 2] (ditto)

2) Some external process has stored in vector **A** these terms for the *Taylor Series Expansion* of some function *f(x)*, in increasing order of *x* powers (as is customary for *Taylor Series* and formal power series in general):

$$f(x) = \frac{1}{2} - \frac{1}{6}x + \frac{1}{1296}x^3 - \frac{1}{933120}x^5 + \frac{1}{1410877440}x^7 + O(x^8)$$

and we need to compute its *smallest real root* but though **MAT..PROOT** does indeed find the smallest roots first, it expects the coefficients to be in *decreasing* order of *x* powers. Enter **MAT A..REV** to the rescue:

```
DESTROY ALL @ OPTION BASE 1 @ FIX 7 END LINE
DIM F(8) @ COMPLEX R(7) @ MAT INPUT F END LINE
F(1) ?
       1/2,-1/6,0,1/1296,0,-1/933120,0,1/1410877440 END LINE
MAT F=REV(F) @ MAT R=PROOT(F) @ DISP R(1) END LINE
```

(3.1415927, 0.0000000) which indeed it's real and looks good.

Rationale: Another basic array function which works synergically with other such functions (**MAT..PROOT** here) to simplify the handling and processing of arrays, while taking very few resources to implement.

REPT / IMPT **Real / Imaginary Parts of Matrix**

```
MAT A=REPT(B)
MAT A=IMPT(B)
```

Where **A** and **B** are arrays.

Assigns to **A** the real parts / imaginary parts of the elements of **B**. Typically **A** will be real and **B** complex.

Not usable in CALC mode.

Example

Split the following complex matrix **A** into its real and imaginary parts, to be stored to real matrices **R** and **C**:

$$\mathbf{A} = \begin{bmatrix} 1+i & 2-i & -3-3i \\ -i & 4 & 6-i \\ 0.5-i & 7+8i & -5 \end{bmatrix}$$

```
DESTROY ALL @ OPTION BASE 1 @ STD [END LINE]
COMPLEX A(3,3) @ DIM R(3,3),C(3,3) [END LINE]
MAT INPUT A [END LINE]
A(1,1)?
        (1,1),(2,-1),(-3,-3),(0,-1),4,(6,-1),(.5,-1),(7,8),-5 [END LINE]
```

```
MAT R=REPT(A) @ MAT DISP R; [END LINE]
```

$$\mathbf{R} = \begin{bmatrix} 1 & 2 & -3 \\ 0 & 4 & 6 \\ 0.5 & 7 & -5 \end{bmatrix}$$

```
MAT C=IMPT(A) @ MAT DISP C; [END LINE]
```

$$\mathbf{C} = \begin{bmatrix} 1 & -1 & -3 \\ -1 & 0 & -1 \\ -1 & 8 & 0 \end{bmatrix}$$

Rationale: Two very basic operation that shouldn't have been left out from the original *Math Pac*, and further it's the only way to quickly get the real/imaginary parts of a complex matrix without using slow *BASIC* loops. A matrix instruction set which forces you to use slow *BASIC* loops to accomplish pretty simple, almost trivial matrix operations is a big *no-no* in my book.

They're fairly easy to implement and thus should take very few resources.

CONJ **Conjugate Matrix**

MAT **A**=CONJ(**B**)

Where **A** and **B** are arrays.

Assigns to **A** the conjugate elements of **B**. If either **A** or **B** are real, only the real parts are assigned.

Not usable in CALC mode.

Examples

1) Replace the elements of the following matrix by their conjugates.

$$\mathbf{A} = \begin{bmatrix} 1+i & 2-i & -3-3i \\ -i & 4 & 6-i \\ 0.5-i & 7+8i & -5 \end{bmatrix}$$

```
DESTROY ALL @ OPTION BASE 1 @ STD END LINE
COMPLEX A(3,3) @ MAT INPUT A END LINE
A(1,1)?
       (1,1),(2,-1),(-3,-3),(0,-1),4,(6,-1),(.5,-1),(7,8),-5 END LINE

MAT A=CONJ(A) @ MAT DISP A; END LINE
```

$$\mathbf{A} = \begin{bmatrix} 1-i & 2+i & -3+3i \\ i & 4 & 6+i \\ 0.5+i & 7-8i & -5 \end{bmatrix}$$

2) Now obtain the matrix transpose of the following matrix:

$$\mathbf{A} = \begin{bmatrix} 6+2i & -i & -4 \\ -2-i & 4i & 2+2i \end{bmatrix}$$

We can't just simply use the existing **MAT..TRN** keyword because it returns the *conjugate transpose*, so we'll first obtain that conjugate but then we'll conjugate the result, thus getting the desired *"unconjugated"* transpose:

```
DESTROY ALL @ OPTION BASE 1 @ STD END LINE
COMPLEX A(2,3) @ MAT INPUT A END LINE
A(1,1)?
       (6,2),(0,-1),-4,(-2,-1),(0,4),(2,2) END LINE

MAT A=TRN(A) @ MAT A=CONJ(A) @ MAT DISP A; END LINE
```

$$\mathbf{A} = \begin{bmatrix} 6+2i & -2-i \\ -i & 4i \\ -4 & 2+2i \end{bmatrix}$$

Rationale: This is a very basic operation that shouldn't have been left out from the *Math Pac* and further it's the only way to quickly get the *actual* transpose of a complex array (and not the *conjugate* one) without using slow *BASIC* loops, as shown above. Also, it's fairly easy to implement and should take few resources.

ABS **Absolute Matrix**

MAT **A**=ABS(**B**)

Where **A** and **B** are arrays.

Assigns to **A** the absolute values of the elements of **B**. Both **A** and **B** can be real or complex but it makes more sense for **A** to be real as the absolute values returned are always nonnegative real numbers.

Not usable in CALC mode.

Example

Assign to the real matrix **R** the absolute values of the elements of the following complex matrix **A**:

$$\mathbf{A} = \begin{bmatrix} 1+i & 2-i & -3-3i \\ -i & 4 & 6-i \\ 0.5-i & 7+8i & -5 \end{bmatrix}$$

```
DESTROY ALL @ OPTION BASE 1 @ FIX 4 [END LINE]
COMPLEX A(3,3) @ DIM R(3,3) [END LINE]
MAT INPUT A [END LINE]
A(1,1)?
       (1,1),(2,-1),(-3,-3),(0,-1),4,(6,-1),(.5,-1),(7,8),-5 [END LINE]

MAT R=ABS(A) @ MAT DISP R; [END LINE]
```

$$\mathbf{R} = \begin{bmatrix} 1.4142 & 2.2361 & 4.2426 \\ 1.0000 & 4.0000 & 6.0828 \\ 1.1180 & 10.6301 & 5.0000 \end{bmatrix}$$

Rationale: A basic operation that takes very little resources to implement but which can greatly help to simplify and significantly accelerate certain important procedures such as a part of solving *nonlinear systems of algebraic equations* and *systems of ordinary differential equations (ODEs)* as well, plus matrix inversion or linear system-solving refinement procedures, among many other such algorithms.

The idea is that you're implementing some iterative process to solve the above, and in every iteration you have to compute an array of *residuals* whose components must be smaller than some threshold to achieve convergence. The **MAT..ABS** keyword can quickly fill up an array with the residuals which you can then check for smallness in a number of ways. And as mentioned, this keyword should take very little resources as the scalar **ABS** keyword alredy exists for both real and complex scalar values.

RAN **Random Matrix**

```
MAT A=RAN[(U,V)]
```

Where **A** is an array and the optional parameters **U** and **V** are scalar variables.

Fills an array with random numbers.

If no optional parameters are given, the random numbers generated will be real numbers between 0 and 1 (the extreme 1 not included). Else they will be integer values between **U** and **V**, both extremes included.

Not usable in CALC mode.

Example

Use seed 1 to create a 3x3 random matrix of real values in [0, 1), then fill a 2x10 matrix with simulated random throws of two dice.

```
DESTROY ALL @ OPTION BASE 1 @ FIX 4 END LINE
DIM A(3,3) @ RANDOMIZE 1 @ MAT A=RAN END LINE
MAT DISP A; END LINE
```

$$\mathbf{A} = \begin{bmatrix} 0.7314 & 0.7721 & 0.9897 \\ 0.2490 & 0.8668 & 0.0343 \\ 0.5882 & 0.0247 & 0.8067 \end{bmatrix}$$

```
DESTROY ALL @ OPTION BASE 1 @ STD END LINE
DIM A(2,10) @ RANDOMIZE 1 @ MAT A=RAN(1,6) END LINE
MAT DISP A; END LINE
```

$$\mathbf{A} = \begin{bmatrix} 6 & 2 & 6 & 1 & 4 & 1 & 5 & 5 & 1 & 1 \\ 1 & 6 & 4 & 4 & 2 & 5 & 3 & 3 & 5 & 1 \end{bmatrix}$$

Rationale: When doing simulations or testing algorithms there's frequently the need to use some random data as initial starting values to see if the algorithm behaves as expected or to seed the first stage of the simulation. Also, having one or several matrices initialized with random values is very common when designing games, so random matrices find frequent use in all sorts of tasks.

Creating and filling up matrices with random values in *BASIC* requires nested loops and is rather clumsy, *RAM*-consuming and slow, so having a **MAT..RAN** keyword that does it in a single statement at assembly-language speeds is very convenient and consistent with the already existing statements **MAT..ZER** (fill up a matrix with 0's), **MAT..IDN** (create an identity matrix) and **MAT..CON** (fill up a matrix with 1's), and now **MAT..RAN** (fill up a matrix with random values) completes the suite.

ROUND **Matrix Rounding**

```
MAT A=ROUND(B [,N])
```

Where **A** and **B** are arrays and the optional parameter **N** is a scalar variable.

Assigns to **A** the values of the elements of **B** rounded to N places. If N is not provided they are rounded to the nearest integer by default (as does the **IROUND** keyword for real scalar values).

N can be any real value $|N| \leq 12$. If N is negative rounding will take place *to the left* of the decimal point (e.g.: *1,234.567* rounded to 2 places would be *1,234.57* but rounded to -3 places it would be *1,000.00*)

Not usable in CALC mode.

Example

Compute the *exact* inverse of the following 3x3 *Hilbert* matrix and return the result to matrix **B**:

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

```
DESTROY ALL @ OPTION BASE 1 @ STD @ DIM A(3,3),B(3,3)
```  **END LINE**

We now fill up the matrix elements automatically (instead of doing it manually using a **MAT..INPUT** statement):

```
FOR I=1 TO 3 @ FOR J=1 TO 3 @ A(I,J)=1/(I+J-1) @ NEXT J @ NEXT I
```  **END LINE**

Checking the inverse's determinant (which is the reciprocal of **A**'s determinant) we get:

```
1/DET(A)
```  **END LINE**

2160.00000009

which is very close to an integer, so probably the inverse's elements should all be integers as well. Let's see:

```
MAT B=INV(A) @ MAT DISP B;
```  **END LINE**

$$\mathbf{B} = \begin{bmatrix} 9.0000000004 & -36.000000002 & 30.0000000018 \\ -36.0000000019 & 192.000000009 & -180.000000008 \\ 30.0000000017 & -180.000000008 & 180.000000007 \end{bmatrix}$$

Close but no cigar. This can be easily remedied by rounding them all to the nearest integer:

```
MAT B=ROUND(B) @ MAT DISP B;
```  **END LINE**

$$\mathbf{B} = \begin{bmatrix} 9 & -36 & 30 \\ -36 & 192 & -180 \\ 30 & -180 & 180 \end{bmatrix}$$

Rationale: Many operations with matrices tend to give results which are either misleadingly precise (such as getting 12-decimal results for 4-digit experimentally-obtained data) or slightly inaccurate (giving non-integer values for theoretically integer results). This happens all the time in science, engineering and numerical analysis and the remedy is to simply round the results (to 4 decimals in the former case, to the nearest integer in the latter), which the **MAT..ROUND** keyword can do quickly with a single statement.

MAT **A**=**B**^(**N**)

Where **A** and **B** are square matrices and **N** is a scalar variable.

Assigns to **A** the matrix **B** raised to the N^{th} power, where N is an integer ≥ 0 (else its integer absolute value will be used instead). Also, $\mathbf{A}^{\mathbf{0}}$ will be returned as the *Identity Matrix* and $\mathbf{A}^{\mathbf{1}}$ as the matrix **A** itself.

Not usable in CALC mode.

Example

Compute $\mathbf{B} = \mathbf{A}^{\mathbf{3}}$ where **A** is the famous *"Luo Shu",* an ancient magic square with magic constant 15, and check that **B** is also a magic square but with magic constant 15^3:

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

```
DESTROY ALL @ OPTION BASE 1 @ STD [END LINE]
DIM A(3,3),B(3,3) @ MAT INPUT A [END LINE]
A(1,1)?
        4,9,2,3,5,7,8,1,6 [END LINE]

MAT B=A^(3) @ MAT DISP B; [END LINE]
```

$$\mathbf{B} = \begin{bmatrix} 1149 & 1029 & 1197 \\ 1173 & 1125 & 1077 \\ 1053 & 1221 & 1101 \end{bmatrix}$$

and now we can check that every row, column and diagonal adds up to $3{,}375 = 15^3$, like this:

```
DIM R(3),C(3) @ MAT R=RSUM(B) @ MAT C=CSUM(B) @ MAT DISP R;C; [END LINE]
```

$$\mathbf{R} = \begin{bmatrix} 3375 \\ 3375 \\ 3375 \end{bmatrix}, \qquad \mathbf{C} = \begin{bmatrix} 3375 \\ 3375 \\ 3375 \end{bmatrix}$$

That takes care of the sums of rows and columns, and for the diagonals we can use the new **TRACE** function:

```
TRACE(B);TRACE(B,1) [END LINE]

        3375       3375
```

so **B** is indeed a *bona fide* magic square with magic constant $3{,}375 = 15^3$.

Rationale: This is again a truly basic, simple matrix arithmetic function that's very easy to implement (just repeated matrix multiplications using a binary decomposition of the integer exponent N to minimize the number of multiplications needed), thus taking little resources and conveniently freeing the user from having to code it as a *BASIC* loop using another matrix. Possible applications are many, both casual use and as part of more elaborate algorithms; for instance it can be used to help evaluate approximations to more sophisticated transcendental matrix functions such as *exp(**A**)* or *sin(**A**)*, say, where **A** is a square matrix.

PCHAR **Characteristic Polynomial**

MAT **A**=PCHAR(**B**)

Where **A** is an array and **B** is a square matrix.

Assigns to **A** the coefficients of the *Characteristic Polynomial* of **B** (which doesn't need to be symmetric).

Not usable in CALC mode.

Example

Compute all eigenvalues, real and complex, of the following matrix **A**:

$$\mathbf{A} = \begin{bmatrix} 5 & 1 & 2 & 0 & 4 \\ 1 & 4 & 2 & 1 & 3 \\ 2 & 2 & 5 & 4 & 0 \\ 0 & 1 & 4 & 1 & 3 \\ 4 & 3 & 0 & 3 & 4 \end{bmatrix}$$

We will first compute the *Characteristic Polynomial* of **A** with a single **MAT..PCHAR** statement, then the eigenvalues will be its roots and will be computed using a **MAT..PROOT** statement, thus just 2 statements in all:

```
DESTROY ALL @ OPTION BASE 1 @ STD END LINE
DIM A(5,5),B(6) @ MAT INPUT A END LINE
A(1,1)?
       5,1,2,0,4,1,4,2,1,3,2,2,5,4,0,0,1,4,1,3,4,3,0,3,4 END LINE
```

```
MAT B=PCHAR(A)@ MAT DISP B; END LINE
```

$$\mathbf{B} = \begin{bmatrix} 1 \\ -19 \\ 79 \\ 146 \\ -1153 \\ 1222 \end{bmatrix}$$

so the *Characteristic Polynomial* of **A** is: $P(x) = x^5 - 19x^4 + 79x^3 + 146x^2 - 1153x + 1222$

and the 5 eigenvalues of **A** are its five roots, which we'll presently compute and display, like this:

```
COMPLEX R(5) @ MAT R=PROOT(B) @ MAT DISP R; END LINE
```

$$\mathbf{R} = \begin{bmatrix} 1.49765770722 + 0i \\ 3.36187557654 + 0i \\ -3.55783865798 + 0i \\ 5.6725513961 + 0i \\ 12.0257539781 - 0i \end{bmatrix}$$, so all 5 eigenvalues are *real*, as their imaginary parts are 0.

Rationale: Computing eigenvalues of a matrix is essential in many important areas of science and engineering as well as analysis of algorithms, etc. This keyword allows for it at assembler speeds with full 12-digit accuracy and in just two statements. It could be done in just one with a specific keyword but the *Characteristic Polynomial* is useful in itself and can be quickly computed exactly with a fast, simple, non-iterative algorithm. Also, if desired the *determinant* and the *inverse* of the original matrix can be easily obtained as side effects of the main computation, with greater accuracy than using **MAT..INV** and **DET**.

PDER **Polynomial Derivatives**

MAT **A**=PDER(**B,N**)

Where **A** and **B** are arrays and **N** is a real variable.

Assigns to **A** the coefficients of the N^{th} derivative of the polynomial whose coefficients are stored in **B**.

N should be an integer ≥ 0, else the absolute value of its integer part will be used. if **N** = 0 the derivative will be the polynomial itself. If **N** > the degree of the polynomial, the N^{th} derivative will have all its coefficients as 0.

Not usable in CALC mode.

Examples

1) Display the coefficients of the 1^{st} , 2^{nd} and 3^{rd} derivatives of the following polynomial:

$$P(x) = 225\ x^4 - 425\ x^3 + 170\ x^2 + 370\ x + 100$$

```
DESTROY ALL @ OPTION BASE 1 @ STD END LINE
DIM P(5),D1(4),D2(3),D3(2) @ MAT INPUT P END LINE
P(1) ?
       225,-425,170,370,100 END LINE
```

```
MAT D1=PDER(P,1)@ MAT D2=PDER(P,2)@ @ MAT D3=PDER(P,3) @ MAT DISP D1;D2;D3 END LINE
```

$$\mathbf{D1} = \begin{bmatrix} 900 \\ -1275 \\ 340 \\ 370 \end{bmatrix}, \quad \mathbf{D2} = \begin{bmatrix} 2700 \\ -2550 \\ 340 \end{bmatrix}, \quad \mathbf{D3} = \begin{bmatrix} 5400 \\ -2550 \end{bmatrix}$$

so we have $P'(x) = 900x^3 - 1275x^2 + 340x + 370$, $P''(x) = 2700x^2 - 2550x + 340$, $P^{(3)}(x) = 5400x - 2550$

2) Find a complex root of $P(z) = (2 + 3i)\ z^3 - (1 + 2i)\ z^2 - (3 + 4i)\ z - (6 + 8i) = 0$ near $z = -(1 + i)$

We'll use five iterations of *Newton's Method*: $z_0 = -(1 + i)$, $z_{n+1} = z_n - P(z) / P'(z)$, like this:

```
DESTROY ALL @ OPTION BASE 1 @ FIX 5 END LINE
COMPLEX P(4),D(3),Z @ MAT INPUT P END LINE
P(1) ?
       (2,3),-(1,2),-(3,4),-(6,8) END LINE
```

```
MAT D=PDER(P,1) @ Z=-(1,1) END LINE
FOR I=1 TO 5 @ Z=Z-PEVAL(P,Z)/PEVAL(D,Z) @ NEXT I END LINE
```

```
        (-0.70473, -0.91388)
```

so the complex root is: $z = -0.70473 - 0.91388\ i$, correct to all digits shown.

Rationale: Finding the derivatives of a polynomial in *exact* form is an essential task and **MAT..PDER** makes it as fast, accurate and simple as possible. The applications are endless, such as finding the extrema and points of inflection of a polynomial (which might be a *Taylor Series* or a fit to some function or even to experimental scientific and engineering data). The example above shows the synergy between **MAT..PDER** and **MAT..PEVAL**, working together to find with utmost ease (from the command line no less !) a *complex* root of a polynomial with *complex* coefficients, something that neither **FNROOT** nor **PROOT** can do.

PFIT **Polynomial Exact Fit**

```
MAT A=PFIT(X,Y)
```

Where **A**, **X** and **Y** are vectors of the same size.

Assigns to **A** the coefficients of the polynomial which exactly fits (passes through) the data points (x_i , y_i) whose coordinates are stored in the **X** and **Y** vectors, respectively.

The abscisas need not be equally spaced and also they don't need to be input in any particular order.

Not usable in CALC mode.

Example

Fit a polynomial to the following data points (x_i , y_i): (2,1) , (5,70) , (3,16) (notice that they aren't equispaced and further they're given in some arbitrary order).

First we'll ask the user to enter the points in the *X*, *Y* vectors, then we'll compute and output the fitting polynomial and finally we'll check that the polynomial does indeed exactly pass through the data points:

```
DESTROY ALL @ OPTION BASE 1 @ STD   END LINE
DIM P(3),X(3),Y(3) @ MAT INPUT X,Y   END LINE
X(1)?
        2,5,3   END LINE
Y(1)?
        1,70,16   END LINE

MAT P=PFIT(X,Y) @ MAT DISP P;   END LINE
```

$$\mathbf{P} = \begin{bmatrix} 4 \\ -5 \\ -5 \end{bmatrix}$$

so the fitting polynomial **P** is: $P(x) = 4\,x^2 - 5\,x - 5$, and we'll check that it passes through the given points :

```
FOR I=1 TO 3 @ DISP X(I);Y(I),PEVAL(P,X(I)) @ NEXT I   END LINE
```

```
 2    1      1
 5   70     70
 3   16     16
```

Rationale: This is another extremely basic but extremely important functionality in all of math, science and engineering, the capability to fit a polynomial to a set of data points, in this case an exact fit where the polynomial does indeed pass through all given points, which is called a *collocation polynomial* in the literature. Once computed it can be evaluated for interpolations, inverse interpolations, integrated, differentiated, its roots and extrema can be found, and so on and so forth.

Computing it is both very fast and needs very few resources as it can use existing functionality in the ***Math Pac 2*** to accomplish the task (mostly calling once the **MAT..CON** and **MAT..SYS** internal routines). Using this new keyword avoids all the drudgery of having to code it in *BASIC* and in particular the need to use two nested, slow loops. Notice in the example above the synergy with another proposed keyword, **PEVAL**, for quick evaluation of the just computed polynomial, which is already in the form that **PEVAL** accepts.

PFITLS **Polynomial Least Squares Fit**

MAT **A**=PFITLS(**X,Y**)

Where **A**, **X** and **Y** are vectors. **X** and **Y** must be same size but **A**'s size is usually (much) smaller.

Assigns to **A** the coefficients of the polynomial which best fits in the *Least Squares* sense the data points (x_i , y_i), whose coordinates are stored in the **X** and **Y** vectors, respectively. The number of data points is the common size of **X** and **Y** while the size of **A** indicates the fitting polynomial's degree.

The abscisas need not be equally spaced and also they don't need to be input in any particular order.

Not usable in CALC mode.

Example

Fit a 2nd -degree *Least Squares* polynomial to the following data points (x_i , y_i):

x	*0*	*1*	*2*	*3*	*4*	*5*
y	*10.2*	*10.9*	*14.3*	*18.9*	*26.3*	*34.8*

First we'll ask the user to enter the points in the *X*, *Y* vectors, then we'll compute and output the fitting polynomial and finally we'll evaluate the polynomial at the given x_i values, comparing with the given y_i values.

```
DESTROY ALL @ OPTION BASE 1 @ FIX 3 [END LINE]
DIM P(3),X(6),Y(6) @ MAT INPUT X,Y  [END LINE]
X(1)?
        0,1,2,3,4,5 [END LINE]
Y(1)?
        10.2,10.9,14.3,18.9,26.3,34.8 [END LINE]
```

```
MAT P=PFITLS(X,Y) @ MAT DISP P; [END LINE]
```

$$\mathbf{P} = \begin{bmatrix} 0.982 \\ 0.055 \\ 10.093 \end{bmatrix}$$

so the fitting *Least-Squares* polynomial is: $P(x) = 0.982\,x^2 + 0.055\,x + 10.093$. To check how well it does:

```
FIX 1 @ FOR I=1 TO 6 @ DISP X(I);PEVAL(P,X(I)) @ NEXT I [END LINE]
```

x	*0*	*1*	*2*	*3*	*4*	*5*
y	*10.2*	*10.9*	*14.3*	*18.9*	*26.3*	*34.8*
P(x)	*10.1*	*11.1*	*14.1*	*19.1*	*26.0*	*34.9*

Rationale: As with **MAT..PFIT** above, the ability to fit a *Least Squares* polynomial to a set of data points is an extremely useful tool, even more so since it minimizes the *squares* of the errors instead of exactly passing through all the points, which might lead to oscillations. Also, this type of fit *smoothes* empirical data noise.

As before, once computed it can be evaluated for interpolations and *inverse* interpolations, integrated, differentiated, its roots and extrema can can be found, etc. Computing it is again very fast and needs very few resources as it can use already existing functionality in the ***Math Pac 2*** to accomplish the task.

PEVAL **Polynomial Evaluation**

Y=PEVAL(**P,X**)

Where **X** and **Y** are scalar variables and **P** is an array with N+1 elements holding the coefficients of **P**, where N is the degree of the polynomial being evaluated.

Evaluates at **X** the polynomial whose coefficients are stored in **P** and returns the result to **Y**.

Not usable in CALC mode.

Example

Evaluate the following polynomial at *x = 1, 2, 3* and *4*:

$$5\,x^6 - 45\,x^5 + 225\,x^4 - 425\,x^3 + 170\,x^2 + 370\,x - 500$$

```
DESTROY ALL @ OPTION BASE 1 @ STD END LINE
DIM P(7) @ MAT INPUT P END LINE
P(1) ?
       5,-45,225,-425,170,370,-500 END LINE
```

```
FOR X=1 TO 4 @ DISP X;PEVAL(P,X) @ NEXT X END LINE
```

```
       1   -200
       2   0
       3   1600
       4   8500
```

Rationale: Polynomial evaluation is one of the most frequently tasks performed in any number of disciplines, from curve fitting to interpolation to root finding to evaluating *Taylor* expansions to whatever, so doing it as quickly and conveniently as possible will greatly enhance many programs and their usability.

The *HP-71 Curve Fitting Pac* does include a *binary subprogram* to do it, but it's limited to degree 19 (can't imagine why) and being a subprogram it's somewhat clumsy to use, plus it computes and returns additional values other than the polynomial's evaluation. On the other hand, the proposed **PEVAL** keyword doesn't need to limit the degree to such low values and, being a function, can be used in expressions (except in the pretty useless *CALC* mode), which is much more convenient (see **MAT..PDER**'s second example above).

Matter of fact, it's so useful and so frequently necessary that apart from this page with its description and examples, it also does appear in the examples for several other keywords in this document, as seen.

Implementing this keyword is just a matter of using *Horner's Scheme* to evaluate the polynomial, which can be done very fast and accurately with just N multiplications and additions (where N is the degree of the polynomial), no need for raising the argument to powers.

DINTEG **Discrete Integration**

S=DINTEG(**D,H**)

Where **S** and **H** are scalar variables and **D** is an array which holds the N+1 equally-spaced discrete data points.

Evaluates the integral pertinent to the N+1 datapoints stored in **D**, which are the y_i ordinates corresponding to equally spaced abscisas x_i (where H is the spacing), and returns the result to **S**. The abscisas aren't needed in the computation so they're not input.

N must be even and ≥ 4 and **H** must be > 0.

Not usable in CALC mode.

Example

Compute the integral corresponding to the following discrete dataset, which was obtained by conducting some measurements.

x	*1.00*	*1.05*	*1.10*	*1.15*	*1.20*	*1.25*	*1.30*
y	*1.00000*	*1.02470*	*1.04881*	*1.07238*	*1.09545*	*1.11803*	*1.14018*

There are 7 data points (so we have N=6,which is even and ≥ 4) spaced by 0.05 (H), thus we proceed like this:

```
DESTROY ALL @ OPTION BASE 0 @ FIX 5 END LINE
DIM D(6) @ MAT INPUT D END LINE
D(0) ?
       1,1.02470,1.04881,1.07238,1.09545,1.11803,1.14018 END LINE

DINTEG(D,0.05) END LINE

       0.32149
```

which is correct to all 5 digits shown.

Rationale: The *Math Pac* features the very poweful, nestable **INTEGRAL** keyword which numerically computes the integral between specified limits of a user-provided function *f(x)* that can be evaluated for arbitrary arguments *x* within the interval of integration. So far so good.

However, there's no provision to compute the integral of a *discrete* set of datapoints which aren't computed by evaluating a known function *f(x)* but rather they're obtained by empirical means, conducting some experiment and taking readings or performing measurements. In this very frequent case in science and engineering you end up with a *discrete* dataset, an array of numbers which are the values of an unknown function (probably not that smooth because of experimental errors and noise, so not easy to accurately fit an analytic function to it) and you need to compute the integral but the existing **INTEGRAL** keyword is of no help in this case.

Enter **DINTEG** *(Discrete Integral)*, which accepts the dataset (only the y_i ordinates are needed) and the spacing and produces the resulting integral very quickly and accurately using just a single, convenient function instead of having to use clumsy, slow *BASIC* code or attempting to fit some *f(x)* to the data, which would probably be a mediocre fit anyway because of the experimental errors and noise mentioned. Besides, it can be coded very efficiently and, unlike **INTEGRAL**, using very few resources

TRACE **Matrix Trace**

Y=TRACE(**A** [,**D**])

Where **A** is a square matrix and **D**, **Y** are scalar variables.

Returns to **Y** the value of the trace of matrix **A**, which is the sum of all the elements in the main diagonal, which runs from $A_{1,1}$ to $A_{N,N}$ where N is the dimension of the matrix.

If the optional parameter **D** is supplied, it returns the sum of the elements of the main diagonal if **D** = 0, or the sum of the elements of the opposite main diagonal if **D** ≠ 0. The opposite main diagonal runs from $A_{1,N}$ to $A_{N,1}$.

Not usable in CALC mode.

Example

Compute the trace of the following matrix **A** and check that it equals the sum of all its eigenvalues:

$$\mathbf{A} = \begin{bmatrix} 5 & 1 & 2 & 0 & 4 \\ 1 & 4 & 2 & 1 & 3 \\ 2 & 2 & 5 & 4 & 0 \\ 0 & 1 & 4 & 1 & 3 \\ 4 & 3 & 0 & 3 & 4 \end{bmatrix}$$

First we'll input the matrix and compute its trace:

```
DESTROY ALL @ OPTION BASE 1 @ STD END LINE
DIM A(5,5),B(6) @ MAT INPUT A END LINE
A(1,1)?
       5,1,2,0,4,1,4,2,1,3,2,2,5,4,0,0,1,4,1,3,4,3,0,3,4 END LINE
```

```
TRACE(A)

       19
```

We now compute the *Characteristic Polynomial* using the new **MAT..PCHAR** statement (see the keyword reference above), and the eigenvalues are its roots which we compute next using a **MAT..PROOT** statement:

```
MAT B=PCHAR(A)@ COMPLEX R(5) @ MAT R=PROOT(B) @ MAT DISP R; END LINE
```

$$\mathbf{R} = \begin{bmatrix} 1.49765770722 + 0i \\ 3.36187557654 + 0i \\ -3.55783865798 + 0i \\ 5.6725513961 + 0i \\ 12.0257539781 - 0i \end{bmatrix}$$, so all 5 eigenvalues are indeed *real*, as their imaginary parts are 0.

and the real part of their sum will be (the imaginary parts are all 0):

```
REPT(SUM(R))

       19
```

Rationale: The trace (**Tr** is the usual name in the literature) is a basic matrix function, which complements the already existing **SUM** functions and further it takes *extremely* few resources to implement, so **TRACE** truly belongs with them. See it in use in the example at the description of the **MAT..^** keyword above.

LAMW **Lambert W Function**

Y=LAMW(**X** [**,B**])

Where **X** and **Y** are scalar variables.

Evaluates the **Lambert W** function for the argument **X** and the branch **B** (0 or -1, default is main branch, B = 0) and returns the result to **Y**.

Can be used in CALC mode.

Examples

Evaluate *Lambert W* function (main branch) for *X = 1, 2, 3* and *4*, both branches for *X = −0.2*, and then solve the equation $x^x = 5$, checking the result.

```
DESTROY ALL @ STD [END LINE]
FOR X=1 TO 4 @ DISP X;LAMW(X) @ NEXT X [END LINE]

        1   .567143290411
        2   .852605502014
        3   1.04990889497
        4   1.2021678732

LAMW(-0.2)@ LAMW(-0.2,-1) [END LINE]

        -.259171101818      the main branch (branch 0)
        -2.54264135777      the other real branch (branch -1)
```

A root of that equation is given by $x = e^{W(\ln(5))}$, computed and then checked like this:

```
EXP(LAMW(LN(5))) [END LINE]

        2.12937248276

RES^RES [END LINE]

        5
```

Rationale: *Lambert W* function, the new kid in town, has become a true sensation since its 'rediscovery' in recent times. Its awesome mathematical properties and its evergrowing number of important applications in all fields of engineering and science in general have put it under the limelight, acclaimed by most every professional in the fields of mathematics and applied sciences, to the point that *it's been hailed and proposed as a new elementary function*, to join the ranks of exponentials, logarithms and trigonometrics.
I for once second that proposal.

However, though every worthy math package includes it (though called *ProductLog* in *Mathematica*) to this day no calculator has a button to compute it, so let's right this wrong and implement it in the ***Math Pac 2***.

AGM **Arithmetic-Geometric Mean**

M=AGM(**X,Y**)

Where **X** , **Y** and **M** are scalar variables.

Evaluates the **Arithmetic-Geometric Mean** for the arguments **X** and **Y** and returns the result to **M**.

Can be used in CALC mode.

Examples

1) Evaluate the *Arithmetic-Geometric Mean* for the following pairs *(x, y): (1,3), (5,7), (π, e).*

```
DESTROY ALL @ STD [END LINE]
AGM(1,3);AGM(5,7);AGM(PI,EXP(1)) [END LINE]

        1.86361678324       5.9579660133        2.92610855157
```

2) Check for *x = 0.71* the following identity: $AGM(1+x, 1-x) = \frac{\pi}{2K(x)}$, where *K(x)* is called the *complete elliptic integral of the first kind*.

$$K(x) = \int_0^{\frac{\pi}{2}} \frac{1}{\sqrt{1 - x^2 sin^2\varphi}}\, d\varphi$$

```
DESTROY ALL @ FIX 11 @ RADIANS [END LINE]
X=0.71 @ AGM(1+X,1-X) [END LINE]
PI/(2*INTEGRAL(0,PI/2,0,1/SQR(1-(X*SIN(IVAR))^2))) [END LINE]

        .84562168101
        .84562168101
```

which completely agree when rounded to 11 digits as shown, thus confirming the identity for this particular case.

Rationale: The *Arithmetic-Geometric Mean (AGM)* function has been the subject of much attention since Euler at the very least, by such mathematical celebrities as Gauss himself. It has a plethora of important applications, among them the fact that it can be used to very quickly and efficiently compute many essential functions such as all the elementary ones, as well as other less known (but extremely important in engineering applications) such as the elliptic functions, as demonstrated in the example above (which can be reversed to compute *K(x)* in terms of the **AGM**), and even to compute π itself at unsurpassed speeds.

As was the case with the *Lambert W* function, to the best of my knowledge no calculator has a button or function to compute the *AGM*, so this is the right time to remedy that sorry state of affairs, methinks.

ROUND **Scalar Rounding**

Y=ROUND(**X** [,**N**])

Where **X**, **Y** and **N** are scalar variables.

Assigns to **Y** the value of **X** rounded to **N** places. If **N** is not provided (or **N** = 0) then **X** will be rounded to the nearest integer by default (as does the **IROUND** keyword).

N can be any real value $|N| \leq 12$, but only its integer part will be used.

If N is negative then rounding will take place *to the left* of the decimal point (e.g.: *1,234.567* rounded to two decimal places would be *1,234.57*, but if rounded to *−3* places then it would be *1,000.00*)

Can be used in CALC mode.

Example

Show the values obtained by rounding $1000000*\pi$ from +6 down to −6 decimal places:

```
DESTROY ALL @ STD END LINE

FOR N=1 TO 6 TO -6 STEP -1 @ DISP N;ROUND(1000000*PI,N) @ NEXT N END LINE
```

```
 6   3141592.65359      rounded to 6 decimal places
 5   3141592.6536       rounded to 5 decimal places
 4   3141592.6536              ...
 3   3141592.654               ...
 4   3141592.65                ...
 1   3141592.7          rounded to 1 decimal place

 0   3141593            rounded to the nearest integer (same as IROUND )

-1   3141590            rounded to the next 10's
-2   3141600            rounded to the next 100's
-3   3142000            rounded to the next 1,000's
-4   3140000                   ...
-5   3100000                   ...
-6   3000000            rounded to the next million
```

Rationale: The original *Math Pac* does include the very useful **IROUND** keyword, which rounds its scalar argument to the nearest integer, but once again *HP* came short for no obvious reason and stopped at that, instead of including a much needed **ROUND** function which would round its arguments to N decimal places.

Such a function is needed all the time, *all the time*, and it's extremely simple to implement, yet *HP* didn't see fit to include it in either the *HP-71 System ROMs* or the original *Math Pac*. Fortunately, we can amend here that sorry mishap and include the **ROUND** keyword in the ***Math Pac 2*** instruction set, at long last !

RNDG **Gaussian Random Number**

X=RNDG[(**M,D**)]

Where **X**, **M** and **D** are scalar variables.

Generates a random number subject to a Gaussian distribution (unlike those generated by the built-in RND function, which follow the uniform distribution) and returns it to **X**, updating the random number seed as well.

If no optional parameters are given, the random numbers generated will have mean 0 and standard deviation 1 by default. Else, they will have mean **M** and standard deviation **D**.

Can be used in CALC mode.

Examples

1) Generate seven random numbers with the default Gaussian distribution starting with the seed **1**.

```
DESTROY ALL @ FIX 6 @ RANDOMIZE 1 END LINE
FOR I=1 TO 7 @ DISP RNDG; @ NEXT I @ DISP END LINE
```

```
 -0.053994  0.080641  -0.029463  0.747547  -0.269262  0.241891  -1.587433
```

2) Generate 1,000 random numbers with the default Gaussian distribution and tally how many fall in each of a set of 12 boxes equally spaced.

```
DESTROY ALL @ OPTION BASE 0 @ RANDOMIZE 1 @ STD END LINE
DIM A(11) @ MAT A=ZER END LINE
FOR I=1 TO 1000 @ N=RNDG+6 @ A(N)=A(N)+1 @ NEXT I END LINE
MAT DISP A; END LINE
```

```
 0  0  1  3  48  289  367  235  52  5  0  0
```

which indeed looks pretty Gaussian.

Note: the **6** is an index offset, necessary as array indexes can't be negative, and further we assumed that with such a small sample (just 1,000 numbers) no generated values would be more than 6 standard deviations from the mean, which was indeed the case as can be ascertained by considering the two zero counts at either extreme.

Rationale: **RND**, the built-in random number generator, only produces numbers *uniformly* distributed but there are many important applications in all fields that do require *Gaussian* distributions because that fits many real-life processes in all areas from biology to finance.

No calculator that I know of includes a Gaussian random number generator so **RNDG** will be a welcome first, and a pretty useful one at that. As for resources required, the values can be generated in several ways depending on the compromise between accuracy and execution time but all of them use very little code.

MODP **Modular Exponentiation**

R=MODP(**A,B,M**)

Where **R**, **A, B** and **M** are real scalar variables dealing with integer values.

Calculates **A^B** mod **M** and returns the value to **R**.

Only the absolute integer parts of **A**, **B** and **M** (all of them ≥ 1) will be used in the computation.

Can be used in CALC mode.

Examples

1) Compute the following: 73^{55} *mod 31*, 23^{391} *mod 55*, 31^{397} *mod 55*.

```
DESTROY ALL @ STD  END LINE
MODP(73,55,31); MODP(23,391,55); MODP(31,397,55)  END LINE
```

```
        26      12      26
```

2) Check whether the following numbers are *composite*: *994787, 974153, 994793*.

As per *Fermat's Little Theorem*, a number *P* is prime *iff* N^{P-1} *mod P = 1* for all N not divisible by *P* . Using just a single value of N, if the result is ≠ 1 then *P* is *definitely* composite. When using several values of N, if all results are 1 then *P* is a *probable* prime (unless *P* is a *Carmichael* number, but these are quite rare). Let's proceed:

```
MODP(2,994786,994787)  END LINE
```

(we're computing 2^{994786} mod 994787)

```
        563329
```

the result is ≠ 1 so 994787 is definitely *composite* (actually 994787 = 29 * 34303).

```
MODP(2,974152,974153)  END LINE
```

```
        46457
```

the result is ≠ 1 so 974153 is definitely *composite* (actually 974153 = 983 * 991)

```
MODP(2,994792,994793)  END LINE
```

```
        1
```

the result is **1** so 994793 is a *probable prime*. Let's try a few large random bases N:

```
RANDOMIZE SQR(5) @ P=994793  END LINE
FOR I=1 TO 5 @ N=2*INT(RND*1E5)+1 @ (N,MODP(N,P-1,P)); @ NEXT I @ DISP  END LINE
```

```
        (104309, 1)  (11879, 1)  (133321, 1)  (55591, 1)  (121587, 1)
```

all results are **1** so *994793* is either *very* probably prime or it is a *Carmichael* number (it isn't, it's a prime).

Rationale: Doing modular arithmetic is rather easy using the **MOD**, **RED** and **RMD** functions but *modular exponentiation* is much more difficult (most definitely for really big exponents like the ones used in the examples above) and this **MODP** keyword provides for it very quickly and efficiently, thus completing the modular capabilities of the *HP-71B*. As for real-world applications, there are many in various fields, such as anything to do with primes, divisibility, cryptography and encoding/decoding.

2. Extensions to Existing Keywords

- Unlike other *HP BASIC* implementations, in the *HP-71 Math Pac* the **DET** (determinant) function is implemented only for *real* square matrices, which can be pretty limiting. It should be extended to also work for *complex* matrices. This will also mean that the **DET** and **DETL** (last determinant) parameterless functions must be able to return the complex value of the last determinant computed.

- The **BVAL** and **BSTR$** keywords in the *Math Pac* are limited to work only with bases **2**, **8** and **16** (besides base 10, of course) for no reason I can fathom. They should be extended to work with *all* bases from **2** (digits *0,1*) to **36** (digits *0-9*, *A-Z*), which are no more difficult to handle than base 16 (digits *0-9* and *A-F*). The algorithms to convert to/from base 10 are pretty straightforward involving just integer arithmetic, but coded in *BASIC* they're slow and a chore so this extension would be very welcome.

3. Additional Aliases

The following additional *aliases* would be useful for the reasons given below. They shouldn't take much resources to implement as they'll simply call the relevant existing routines for parsing, execution, etc.

- **SOLVE** as an *alias* for **FNROOT**

  Rationale: **FNROOT** is quite verbose (6 characters) and it's an unnatural name. Almost every *HP* calculator (or other brands) that do have a *Solve* function call it just that, ***Solve***. **FNROOT** is difficult to remember if only occasionally used, and as it begins with **FN** it can be confused with *user-defined function invocations*, which also begin with **FN** (e.g.: `Y=FNR(X)`). The **SOLVE** alias would alleviate all that.

- **INTG** (or **INTEG*I***) as an *alias* for **INTEGRAL**

  Rationale: Ever since I bought the pretty-expensive original *Math Pac* in 1984 I thought that **INTEGRAL** was exceedingly *verbose* (8 characters, to cry out loud !!). It takes long to key it in (and usually you do it wrongly first time), and with the puny 22-char display it takes more than 1/3rd of the display by itself, leaving just 14 characters to add and see the other 3 arguments (including the function to integrate), thus forcing you to slowly scroll the display back and forth to check the correctness of the expression before committing to an usually lengthy computation. A real chore.

  Further, to add insult to injury, if you nest several integrals, which the *Math Pac* does allow, you absolutely run out of room in the display and even in the display buffer. For instance, a triple integral already takes 30 characters just for the **INTEGRAL**`(... `**INTEGRAL**` (... `**INTEGRAL**`( ...)))` invocation itself. Ridiculous and frustrating. Adding the much shorter *alias* would immensely alleviate that.

4. Low-priority Additional Functions

The following functions should be considered *very low-priority*, to be included *only* after higher-priority functionalities have been included already, and only if there's still enough room left for them without exceeding the 32K maximum *ROM* size:

SEC	secant
CSC	cosecant
COT	cotangent

Rationale: These functions can be trivially computed as the reciprocal of the existing trigonometric functions and they aren't much used in programs or manual computations. Assuming they collectively take,

say, 50-100 bytes or more to include them in the *ROM* (parsing, execution, etc.) those bytes would be better used to implement some much more needed and useful functionalities. Only when there's nothing more worthy to include should they be considered, if at all. See my remarks on *compatibility* vs. *functionality*.

The same probably applies to the **`MAT..LUFACT`** statement if it takes significant resources to implement. As with the above functions, it isn't much use, matter of fact it's rarely used, and the resources it takes would be more profitably allocated to some more useful and frequently used functionalities, and the same goes for the **`MAT..INV*`** statement, which in the ***Math Pac 2*** is but an alias for **`MAT..SYS`** and it's only included for compatibility reasons.

I am fully aware that **J-F** is very keen on including such functions for *compatibility* with the *HP-75*, *Series 80* and other *Rocky Mountain BASIC HP* models but I think that if doing so conflicts with including in the ***Math Pac 2*** the most useful functions, then those compatibility functions must go.

The ***Math Pac 2*** will be most useful if it expands the advanced math capabilities of the original *Math Pac*, not as a repository of other old *HP* models' ancillary functions which, if desired, can always be made available in a dedicated *COMPATLEX*, say, which would include all of them for those people interested, no need to waste valuable space in the ***Math Pac 2*** *ROM* for their sake.

IMHO, the goal and main priority must be first and foremost to try and maximally enhance *present functionality*, not to go for unneeded *partial compatibility* with *past* models. Let's not burden the ***Math Pac 2*** with it.

5. Conclusion

Thanks to **Jean-François Garnier** for committing his valuable time and efforts to this magnificent project, and I for one hope he'll find the contents of this document useful and helpful towards its successful completion.

Here's hoping for a most successful outcome for this awesome project. It will probably take several years to come to fruition but it will certainly be worth the wait.