Long Live the HP-15C !

Valentín Albillo (Ex-PPC #4747, HPCC #1075)

According to Wlodek's superb *"Guide to HP Handheld Calculators and Computers"*, the HP-15C was released in July, 1982, and it was meant to be the top scientific model of the Voyager series, looking like the earlier HP-11C, its able sibling, but featuring vastly improved capabilities. The HP-15C's 20th anniversary has recently passed by, and some words commemorating the event are long overdue, thus this article.

Matter of fact, the HP-15C was an incredibly advanced calculator for its time, and arguably it's one of the best calculators ever produced by HP. The fact that so many advanced features could be packaged in such a clever way makes it absolutely plain that lots of pretty clever thinking were applied to its design. Everything, from the physical characteristics of the Voyager series (*ultra-low power drain, solid feel, best keyboard, clearest display, most pocketable*), to the ultra-accurate algorithms implemented (*theoretically best for all arithmetic functions, best practically feasible for the rest*), to the function set *(all logical tests, trascendental functions including hyperbolics even for complex arguments, flags, full statistics, gamma, combinations, permutations, random numbers*), plus cutting-edge advanced capabilities (*complex numbers perfectly integrated down to a full complex RPN stack, matrix operations with multiple matrices and dynamic memory allocation, Solve, Integrate*). The whole set was unbelievable at the time, and still is !

Now, let's say something about each characteristic outlined above, not in an exhaustive style (that would take a whole book !), but pointing out specific facts.

First of all, ***the keyboard layout***. It's really amazing that the designers could fit all the many extra functions and features *in the same number of keys as the HP-11C*, while keeping just the [**f**] and [**g**] prefix keys. A lot of creative thinking was called for, such as using a single **TEST n** instruction *in lieu* of ten logical tests, thus freeing 9 keyboard positions. Next, many keyboard positions were simply reused, by overloading them with new functionalities. For example, the [**+**] key isn't now limited to just adding numbers, but can also add complex values and matrices as well. Other operations are likewise extended, either naturally (the [**÷**] key solving a system of linear equations when applied to matrices), or not-so-naturally (the **Cx,y** and **Px,y** computing combinations and permutations respectively if the arguments are *integers*, else performing some complex transformations if the arguments are *matrices*). There are many instances of smart design all over the keyboard: multiple uses of [**I**] and **(i)**, **STO** and **RCL** admit lots of arguments, operations and functionality (think of

"user" STO, for instance, or RCL arithmetic). Never before had so much thought been given to a small calculator's keyboard layout.

Secondly, ***the superb numerical algorithms***. They are absolutely top-of-the-line, theoretically researched and designed by W. Kahan, a first-magnitude authority on the matter, who had already worked in other HP models such as the HP-91 and the HP-34C. For the HP-15C he excelled himself, and the results are unique, boasting the maximum realizable precision for every operation and function, whether arithmetic, transcendental, complex or matricial. The HP-15C's algorithms were used as the basis for the HP-41C Advantage ROM, the HP-71C Math ROM, the HP-28S, and the HP-42S, all the way up to the latest RPL models.

Thirdly, ***the instruction set*** is astonishingly comprehensive, including every function and feature of practical interest, most of them defined even for complex arguments. Amazingly, its capabilities are even *more* complete than those of the much larger (and *much* more expensive !) HP-71B, even with the Math ROM plugged-in. For instance, you can use **SIN** with complex arguments in both the HP-15C and the HP-71B+Math ROM. But you *can't* use the inverse function **SIN^{-1}** for *complex* arguments with the 71B+Math ROM ! Same for the rest of inverse trigonometric and hyperbolic functions. The HP-15C has no such limitation. The integration of complex numbers with the function set is so thorough that it even features a complete, dynamically allocated RPN stack for complex numbers, **LAST X** and all. The result: working with complex values in RPN as easily as with real values.

And last but not least, ***the programming features and advanced capabilities***. Not only does it include double the memory of the HP-11C, but you have lots of new features that tremendouly increase its power. Like 10 flags, *all 12 conditionals*, and *recall arithmetic*. Like dynamic memory allocation and being able to use **DSZ** and **ISG** on registers other than the I register. Like using *matrix descriptors* and a *parallel stack* so that you can have up to 5 matrices and/or complex numbers on the stack at once. Like the User keyboard and *"user"* functions, which automatically increment the row/column indexes of matrices so that you can input/output their elements using just *one* **STO**/**RCL** instruction per element, with the added program-mode capability to automatically perform a logical test and terminate the loop if the last element has been processed, all with a *single* **STO** or **RCL** ! (this test capability is also featured in other advanced functions, including **SOLVE** and **INTEG**). Add to that a comprehensive set of matrix operations, including both scalar and matrix arithmetic, assignment, transpose, inverse, *system solving*, determinant, norms, residuals, transformations, and even the possibility of working with *complex* matrices. How's that for a most complete feature set ! Let's put it to work !

A sample program: computing up to 208 decimal digits of *e*

Here is a small HP-15C program that I wrote specifically for this article, to give a glimpse of its programming features. This 64-step program will compute ***from 8 to 208 decimal places of Euler's constant***, the well-known transcendental number ***e*** = *2.71828*+ . It is by no means optimized for performance but tries instead to be as short and straightforward as possible. Although you *can* compute more decimal places in an HP-15C, for the purposes of this article this simpler program will do nicely.

Program listing

- **Very Important**: steps 30 and 43 ***must*** be entered ***in*** USER mode, while all the rest ***must*** be entered ***out*** of USER mode. An **u**-like character must appear next to the step number *only* for steps 30 and 43, and no others.

```
01  LBL A          23  RCL A          45  FRAC
02  MATRIX 0       24  X=0?           46  CHS
03  MATRIX 1       25  ISG 2          47  RTN
04    1            26  LBL 0          48  LBL 5
05  DIM A          27  RCL A          49  FRAC
06  DIM B          28  RCL÷ I         50  RCL B
07  RESULT B       29  INT            51  INT
08  STO 2          30u STO A          52  RCL RAN#
09  STO I          31  GTO 2          53    *
10   EEX           32  RCL I          54    +
11    8            33  PSE            55   R/S
12  STO A          34  RCL MATRIX A   56  GTO 4
13   1/X           35  MATRIX 7       57  LBL 2
14  STO RAN#       36  TEST 0         58  RCL* I
15  LBL 1          37  GTO 1          59    -
16  RCL MATRIX A   38  RCL MATRIX B   60  RCL RAN#
17  RCL MATRIX B   39  RCL RAN#       61    ÷
18    +            40    *            62  STO+ A
19  RCL 2          41  FIX 8          63  RCL A
20  STO O          42  LBL 4          64  GTO 0
21  ISG I          43u RCL B
22  LBL 3          44  GTO 5
```

Data Register usage

```
RI   current divisor (2, 3, ...)
R00  row index for matrix elements
R01  column index for matrix elements
```

```
R02  index of first non-zero block
```

Notes

- The constant ***e*** is computed using the well-known formula:

  $$e = 1 + 1/1! + 1/2! + 1/3! + 1/4! + \ldots$$

  using enough terms to achieve the desired precision. We use the powerful matrix capabilities of the HP-15C, holding the current term in a vector (*matrix A*), and the running sum in another vector (*matrix B*). Steps 01-14 dimension and initialize both matrices, as well as some indexes and ancillary constants.

- Starting with 1, each term is computed by dividing the previous one by the corresponding divisor up to a maximum of 208 decimal digits, until we arrive at a term that is zero to the specified precision. This *multiprecision arithmetic* is done by considering each term as composed of N blocks (1-26), each holding *8 digits*, the extra digits in each register being carried out to the next block. Since we may use divisors up to 125, each block is limited to 8 digits, lest the carry would make the next block larger than the 10 digits an HP-15C storage register can hold.

- The addition of each multi-block term to the running total is done using *matrix arithmetic* in steps 16-18. Steps 19-25 increment the divisor and keep track of the first non-zero block of each term to optimize speed by avoiding unnecesary operations. Steps 26-31 and 57-64 perform the multiprecision division.

- The process ends when the current term has all its blocks equal to zero. As all blocks always hold positive values, we can use an advanced matrix operation, the ***Row Norm***, to test for finalization, because in this case the Row Norm will equal zero *if and only if* all block values are *zero*. This is tested in steps 34-37.

- Once this condition is met, the result matrix which holds the running sum is scaled down for display using *matrix-scalar arithmetic* (steps 38-40). The computed answer is then displayed by recalling each block in turn, adding the carry from the previous block, and marking the last one negative (steps 41-56).

- As you can see, though simple, this program does use some of the HP-15C's advanced programming capabilities (as well as a trick or two), including basic matrix operations (**MATRIX 1**), advanced matrix functions (**MATRIX 7**), recall arithmetic (**RCL÷ I**), using registers as indexes including increment-and-test

operations (**ISG 2**) , auto-increment-test-and-loop matrix element access (**uSTO A, uRCL B**), matrix arithmetic (steps 18 and 40), etc.

- Among the tricks, the use of the *fast, 1-byte instructions* **STO RAN#** and **RCL RAN#,** demonstrating a way to store/recall a positive constant less than 1 *without using any regular data registers*. As long as you don't use random numbers (or you don't mind changing the seed at that point) this trick can save you a full register, as well as being faster than having the constant repeated in your program code.

  Caveat emptor: It only works for numbers in [0..1). Try other values (Pi, for instance) and see what happens on recall. This can be used for some pretty efficient 10-power scaling)

Usage instructions

1) After keying in the program, make sure you're *not* in complex mode (press **CF 8** if in doubt), then you must commit enough storage registers to the common pool for the matrix operations. To that effect, press:

   2, **f DIM (i)**

2) Now, enter the number of 8-digit blocks you want to use, from 1 (8 digits) to 26 (208 digits). For example, if you want to compute 200 decimal digits of ***e***, you must specify 200/8 = 25 blocks. Start the program by pressing either

   f PRGM, R/S or **GSB A** or (*in User mode*) **A**

   While running, it will briefly show each succesive divisor used (*2, 3, ...*), then once the computation is over, it will display each block of 8 decimal digits (with an initial *"0."*, in order to preserve leading zeros at the left end of the block), starting with decimals 1st-8th. The very last block will be marked *negative*, to signal the end of the output.

   Let's see an example: to compute the first 24 decimal digits of ***e***, we specify 24/8 = 3 blocks, and proceed like this (in USER mode):

```
3, A   →   (2.00000000)   [divisor = 2]
                ...       [after 2'26"]
       →  (25.00000000)   [divisor = 25]
       →    0.71828182    [decimals  1st- 8th]
```

```
R/S  →    0.84590452     [decimals  9th-16th]
R/S  →   -0.35360274     [decimals 17th-24th]
```

so, after adding a *"2."* at the front and writing down all 8-digit blocks (*that is, minus the initial "0." or "-0."*) in their proper order, we finally get:

$$e = 2.71828182\ 84590452\ 35360274$$

where, due to the accumulation of rounding errors during the process, the last block of the computed answer comes out as *"3536 0274"* while correct is *"3536 0287"*, so we have an error of 13 *ulps* (***u**nits in the **l**ast **p**lace*).

3) Should you need to display the output again, press: **GSB 4**

Assorted results

# blocks	# decimals	Max. divisor	Time	Last block	Error (ulps)
1	8	12	40"	71828178	-4
3	24	25	2'26"	35360274	-13
5	40	35	4'43"	24977552	-20
13	104	73	19'55"	42742712	-34
26	208	125	62'43"	15738281	-60

As you can see, you can compute 100 decimal digits of ***e*** in your HP-15C in under 20 min., and 200 in under one hour. For 208 decimal digits, the final result is:

```
e = 2.71828182 84590452 35360287 47135266 24977572
    47093699 95957496 69676277 24076630 35354759
    45713821 78525166 42742746 63919320 03059921
    81741359 66290435 72900334 29526059 56307381
    32328627 94349076 32338298 80753195 25101901
    15738281
```

The last block of the computed answer comes out as *"1573 8281"* while correct is *"1573 8341"*, so the error is 60 ulps, and thus we've got 206 decimals fully correct.

Final remarks

Well, I certainly hope this commemorative article has revived fond memories of the HP-15C and increased your appreciation of this finest HP-model. I'll consider myself satisfied if you'll contemplate the HP-15C in a different perspective from now on, as the technical marvel and supreme engineering *tour-de-force* it actually is.