HP-71B Fantastic FOUR

Valentín Albillo (#1075, PPC #4747)

Among the many powerful mathematical capabilities provided by the HP-71B Math ROM's comprehensive set of keywords, **`FOUR`** (**Fast Fourier Transform**, FFT for short) must rank as one of the *most* specialized, and thus *less* used and understood of the whole set. After all, lots of people find matrix operations useful in many different fields, not necessarily math-related, and complex numbers also enjoy a wide range of uses, from electrical engineering to surveying. Same goes for numerical integration, root finding, etc. Even exotic special functions such as Gamma and hyperbolics have their share of uses in engineering, architecture, and statistics, but who *needs* to do an FFT or two, let alone *three* ?

It's the purpose of this article to introduce a powerful technique where we'll need to use **`FOUR`** ... well, not four but *three* times. In a row. To be precise, we shall do two Discrete Fourier Transforms, plus one *Inverse* Discrete Fourier Transform, all of which, fortunately, can be done using **`FOUR`**. And now you might be wondering what's that all-too-bizarre application that requires three Fast Fourier Transforms to get accomplished ?

Multiplying numbers. Fast !

That's right, you read correctly. Unlikely and almost unbelievable as it seems, the fastest known way (2005) to multiply two sufficiently large, high-precision numbers, is by means of FFTs. The usual, school method to multiply two numbers goes like this:

```
     53   1st number
 x   12   2nd number
 ------
    106   partial product
    53    partial product
 ------
    636   final result
```

and while this simple algorithm is extremely straightforward to implement in a computer program, even for large numbers, the problem is, the execution time grows *quadratically* with the size of the numbers being multiplied, i.e., for **N**-digit numbers, it is approximately proportional to $\mathbf{N^2}$. Thus, if you *double* the number of digits, the time *quadruples*. This may not seem much of a problem but for certain applications, such as in Experimental Mathematics and Numerical Research, where computations are carried routinely to thousands and even millions of digits, an $\mathbf{O(N^2)}$ time might not be acceptable. Consider for example modern computations of Pi to hundreds and thousands of millions of digits. As all computations of transcendental functions and even divisions can ultimately be reduced to multiplications, the time to *multiply* two large numbers is the *key* to *every* multiprecision computation. Having a quadratically growing multiplication time when dealing with numbers up to a billion (European flavour) digits in size means the computation is usually *unfeasible*.

Can multiplication be done in less than **O(N^2)** ? Yes, there are clever ways to reduce the exponent 2 to lesser values (Karatsuba), arranging operations to use binary shifts to compute subproducts (instead of actual multiplications), and applying such techniques recursively. But even so, we can't get much below exponent 1.58, and that's still *polynomial* time. What we need is *linear* time (exponent 1) or as near to it as possible. This will mean that doubling the number of digits will result in double computing time, not quadruple. Increasing the number of digits a thousand times will increase the computing time a thousand times or so, not a *million* times. Fair enough, but how ? Enter the FFT.

FFT-based multiplication

While the standard and Karatsuba methods of multiplication are perfectly suitable for low decimal digit precision, higher levels of precision require the *much faster* performance achieved by employing a ***convolution-FFT*** approach. Assuming we wish to multiply two multiprecision values **A** and **B**, we consider both as vectors, where each element stores a fixed number of digits of each original value:

$$\mathbf{A} = (a_0, a_1, a_2, \ldots, a_{n-1}), \quad \mathbf{B} = (b_0, b_1, b_2, b_{n-1})$$

then the product of the original numbers, except for releasing carries, is the *acyclic convolution* of the vectors, which can be reduced to *cyclic* convolutions by simply extending both to length 2N by appending N zeroes to each. Thus, the 2N-long product **C** of **A** and **B** is the *cyclic convolution* of vectors **A** and **B**:

$$c_k = \sum_{j=0}^{2n-1} a_j b_{k-j}, \qquad 0 \le k < 2n.$$

and this cyclic convolution is evaluated using FFTs. The results from the final FFT are floating-point numbers which we need to *round* to the nearest integer. A final release of carries gives us the desired multiplication result.

The Discrete and Inverse Discrete Fourier Transforms (DFT) are defined for a complex vector Z:

$$\mathbf{Z} = (z_0, z_1, z_2, \ldots, z_{n-1})$$

as follows:

$$F_k(z) = \sum_{j=0}^{n-1} z_j e^{-2\pi i jk/n} \qquad 0 \le k < n$$

$$F_k^{-1}(z) = \frac{1}{n}\sum_{j=0}^{n-1} z_j e^{2\pi i jk/n} \qquad 0 \le k < n.$$

and the cyclic convolution featured above can be calculated using these DFT as:

$$c_k = F_k^{-1}[F_j(a)F_j(b)], \qquad 0 \le k < 2n.$$

where the three DFT themselves can be efficiently calculated by employing FFTs and that's exactly what the Math ROM's **FOUR** keyword does. Actually, the more you look at it the less believable it seems that you can multiply two *integer* values by computing sums of products of *complex* values times *exponentials* of complex values (which involves *complex floating point* values where we originally had *real integer* values), let alone do it fast !. But the FFT algorithm is **O(N*log(N))**, and thus, for large enough values of **N**, *much faster* than any **O(N^2)** or **O($N^{1.58}$)** algorithm, however simple or efficiently implemented.

The Fantastic FOUR to the rescue !

The fact that the Math ROM includes the **FOUR** keyword makes our implementation of FFT-based multiplication that much *easier* and *efficient*. No need to deal with tricky exponentials of complex values and all that drudgery, **FOUR** will take care of it for us, a *single* keyword replacing *many* lines of intrincate code and what's better, working at assembly language speeds and with internal *extended* precision. This is absolutely vital if we're going to improve on the simpler algorithms and if our final results are to be exact. Remember we're dealing here with *approximate* floating-point values that will get ultimately *rounded*, so extended precision while computing the FFTs is mandatory. In short, **FOUR** gives us simultaneously:

- *Economy*: a single **FOUR** keyword will compute a very complicated FFT, which can be used to compute both a direct DFT and an Inverse DFT.
- *Speed*: **FOUR** is a highly-optimized assembly language implementation of the state-of-the-art Cooley-Tukey's FFT algorithm
- *Accuracy*: **FOUR** works at full internal precision, with extended mantissas and exponent range, plus full provision for denormalization if needed.

This extremely fortunate combination of desirable characteristics will allow us to design a *very efficient* FFT-based multiplication implementation that beats the straightforward methods hands down. Also, for maximum convenience, our implementation will be in the form of a *subprogram* which will encapsulate the whole FFT-based multiplication algorithm without disturbing the caller's environment, and which can be called from other programs or even right from the keyboard if desired.

The subprogram FFTMULT

FFTMULT is a short (*366 bytes*) subprogram (10 lines as formatted for this article but can be reduced to just 8 by simply fitting more statements in each line), which will accept as parameters three vectors, the 1st and 2nd containing the multiprecision integer numbers to be multiplied, and the 3rd vector being where their resulting product will be returned. Here's the listing (the line numbers are completely *arbitrary* as the subprogram uses no line addressing whatsoever):

```
900 SUB FFTMULT(A(),B(),C()) @ I=UBND(A,1) @ J=UBND(B,1)
910 K=I+J @ COMPLEX X(I),Y(J) @ MAT X=A @ MAT Y=B
920 N=LOG2(MAX(I,J)) @ E=10000 @ N=2^(INT(N)+(FP(N)#0))
930 M=2*N @ COMPLEX X(M),Y(M),Z(M) @ MAT X=FOUR(X)
940 MAT Y=FOUR(Y) @ FOR I=1 TO M @ Z(I)=X(I)*Y(I) @ NEXT I
950 COMPLEX Z(M,1) @ MAT Z=TRN(Z) @ MAT Z=FOUR(Z)
960 MAT Z=TRN(Z) @ MAT Z=(M*M)*Z @ COMPLEX Z(M)
970 FOR I=1 TO K @ C(I)=IROUND(REPT(Z(I))) @ NEXT I
980 FOR I=K-1 TO 1 STEP -1 @ J=I+1 @ C(J)=C(J)+MOD(C(I),E)
990 C(I)=C(I) DIV E+C(J) DIV E @ C(J)=MOD(C(J),E) @ NEXT I
```

Notes:

- Each multiprecision number is to be stored divided into *4-digit groups*, each group taking one element of the corresponding vector. Their product is returned in the result vector in a like fashion. The input vectors can have any number of elements but the result vector must have at least the same number of elements as both combined (say 100, 80, and 180). Maximum speed and efficiency are achieved when the number of elements is *an exact power of two* (say 64 elements = 64*4 = 256 digits). For instance, to compute:

  54761407 * 86132724 = 4716749154982668

  you'd do the following from the keyboard:

```
>OPTION BASE 1 @ DIM X(2),Y(2),Z(4)     [ENTER]
>MAT INPUT X,Y                          [ENTER]
X(1)? 5476,1407,8613,2724               [ENTER]
>CALL FFTMULT(X,Y,Z) @ MAT DISP Z       [ENTER]
     4716      7491      5498      2668
```

  Of course, for larger numbers or multiple operations you'd better call **FFTMULT** *programmatically* from a suitable program that sets up the input data and later uses or prints the results. Such a program is used below.

- Though the size of both the input and output vectors is arbitrary, the FFT algorithm internally implemented by **FOUR** requires a size which is *an exact power of 2* (i.e.: 1, 2, 4, 8, 16, etc). **FFTMULT** takes this into account and, for the sake of convenience and generality, internally extends the input/output vectors to meet this requirement. Nevertheless to achieve maximum efficiency original numbers should have 4, 8, 16, 32, ..., 256, 512, ... digits.

Detailed explanation:

As the subprogram is so short, here's a detailed explanation of its inner workings, just in case you'd like to adapt it to some other HP model:

```
900 SUB FFTMULT(A(),B(),C()) @ I=UBND(A,1) @ J=UBND(B,1)
```

The subprogram's header just accepts all three vectors and finds out the size of the input vectors **A** and **B**.

```
910 K=I+J @ COMPLEX X(I),Y(J) @ MAT X=A @ MAT Y=B
```

The size of the result vector is computed as well, and as **FOUR** only works with complex vectors, two such **X**, **Y** are created to stand for the original ones, their elements being copied into **X**'s and **Y**'s real parts.

```
920 N=LOG2(MAX(I,J)) @ E=10000 @ N=2^(INT(N)+(FP(N)#0))
```

FOUR only works with vector sizes equal to an exact power of 2, so a new size is computed for both vectors, equal to the nearest power of 2. Also, a constant is defined to help in post-processing, when releasing the carries.

```
930 M=2*N @ COMPLEX X(M),Y(M),Z(M) @ MAT X=FOUR(X)
```

The new size for the result vector is computed, then both **X** and **Y** are redimensioned to their new common size, while a new complex vector **Z** is created to hold the result. The 1st FFT is then performed on **X**, *in place*.

```
940 MAT Y=FOUR(Y) @ FOR I=1 TO M @ Z(I)=X(I)*Y(I) @ NEXT I
```

Likewise, the 2nd FFT is performed on **Y**, and the corresponding elements of the resulting transformed **X** and **Y** are multiplied together.

```
950 COMPLEX Z(M,1) @ MAT Z=TRN(Z) @ MAT Z=FOUR(Z)
```

Now we need to do the final *inverse* DFT to get the desired product. To that effect we need the *complex conjugate* of **Z**, and this is achieved by redimensioning **Z** from a column vector to a row vector, then using the **TRN** (Matrix Transpose) Math ROM keyword, which, when applied to complex matrices, does *not* return the usual transpose, but the *complex* *conjugate transpose*, i.e.: the elements are not only transposed but replaced by their complex conjugates as well, avoiding a slow user-code loop here. Once we have the complex conjugate of **Z**, the 3rd FFT is performed on it.

```
960 MAT Z=TRN(Z) @ MAT Z=(M*M)*Z @ COMPLEX Z(M)
```

Again, we need the complex conjugate of the transformed **Z**. This is achieved by using **TRN** a second time. Then an scalar-matrix multiplication completes the inverse DFT. The complex vector **Z** now holds the desired product, we merely redimension it back to a one-dimensional vector (for easier, faster indexing) and proceed directly to post-processing.

```
970 FOR I=1 TO K @ C(I)=IROUND(REPT(Z(I))) @ NEXT I
```

Z holds *approximate*, *floating-point* values in both its real and imaginary parts. We're only interested in the *integer*, rounded values of the *real* parts, so we use the handy Math ROM's keywords **REPT** and **IROUND** to extract such values and store them in their final destination, real vector **C**.

```
980 FOR I=K-1 TO 1 STEP -1 @ J=I+1 @ C(J)=C(J)+MOD(C(I),E)
990 C(I)=C(I) DIV E+C(J) DIV E @ C(J)=MOD(C(J),E) @ NEXT I
```

It's nearly over. All we need do is *release the carries*. This loop does exactly that, in place. The product is returned in vector **C** and we're done !

Testing time !

Let's see **FFTMULT** in action ! In order to test the exactness of its results and its speed, we need to enter *very large numbers*, which is a real chore and error-prone.

Instead, we'll use this small (*267 bytes*) 'tester' program, which will generate some very large test values randomly (in a repeatable way), and will provide adequate printing features, in order to display/print both the generated input values as well as their product:

```
100 DESTROY ALL @ OPTION BASE 1 @ STD @ INPUT "N="; N
110 DIM X(N),Y(N),Z(2*N) @ GOSUB 140 @ CALL PM(X) @ DISP "x"
120 CALL PM(Y) @ CALL FFTMULT(X,Y,Z) @ DISP "=" @ CALL PM(Z)
130 END
140 RANDOMIZE PI @ FOR I=1 TO N @ X(I)=INT(RND*10000)
150 Y(I)=INT(RND*10000) @ NEXT I @ RETURN
160 !
170 SUB PM(X()) @ N=16 @ FOR I=1 TO UBND(X,1)
180 DISP USING "#,4Z";X(I) @ J=J+1 @ IF J=N THEN J=0 @ DISP
190 NEXT I @ DISP @ END SUB
```

All it does is set up some initial conditions and ask how many 4-digit groups the generated number will consist of (say, 64 for a 64*4 = 256-digit number), then the required vectors **X**, **Y**, **Z** are dimensioned to their proper sizes, and a call is made to subroutine 140 which simply fills up **X** and **Y** with 4-digit random values. Now **X** and **Y** hold the representation of the large random integers we're going to multiply, and subprogram **PM** is called twice, which simply displays the digits stored in the vectors in an orderly fashion, with zeroes interspersed when necessary. Then **FFTMULT** is called to compute their product, and **PM** is called a final time to display the result. Let's give it a try:

```
>RUN  [ENTER]
N=64  [ENTER]
5476140724498765145164255055853905531677983738245287518763857813
6327817291924274284391508535323487727034799658219644909081309067
2594828012506046508178728897837083606796788063811574831566277675
2690083042313089454529238719564594664469582430009810470529085361
x
8613272483920715414162300723618786418866092062249377500985773606
4662458631705546812446590769217075964055392578377430132675516521
2090736216254127215458485945248165480236637192643292312151263003
6019700443135076274093253344852677580447279631645164021852716383
=
4716749222040286496749070068312320467962230019683526396980208929
8144798698912068898209629755548948239606320386278774464309942475
6421364116184973206557156812839712328524690661728138363939626113
9758747743298173466811693657744732258337534982396879492806571297
3710834368617759391402653484418328618757195082961982876828493898
4603039652792831669763120962090875778261102590570752098488793116
2448856828704962797762336439900426048471952149128458059774536742
2290065181199484453117984492752773390888344738900086346330169263
```

Have a look at these timings for diverse runs of the tester program (calling **FFTMULT**, which uses 4-digit groups) versus a straightforward implementation of the 'school' algorithm (using 6-digit groups for maximum efficiency):

# GROUPS	DIGITS X,Y	DIGITS X*Y	TIME FFT	TIME SCHOOL
16	64	128	24.11	27.51
32	128	256	53.41	62.90
64	256	512	116.74	270.61
128	512	1024	269.89	1034.20

As you can see, the times for the FFT-based algorithm increase approximately *linearly* with the size of the numbers, so that going from 256-digit numbers to 512-digit numbers (a 2x size increase) results in a 2.3x increase in running time, and even going from 128-d numbers to 512-d numbers (a 4x size increase) only means a 5.1x increase. In contrast, the ‘school’ algorithm takes 3.8x and 16.4x longer, respectively, so the *quadratically* growing running time is taking its toll.

Summary

As a *proof-of-concept*, didactic piece of code, **FFTMULT** delivers the goods. It shows how such complex algorithms as FFT-based multiplication do actually *work* and can be implemented in the HP-71B efficiently and even with utterly surprising *ease*, thanks to the powerful Math ROM features. What next ?

Well, being didactic in nature, I refrained from optimizing **FFTMULT** to production quality, sacrificing efficiency and speed for clarity and brevity. Possible improvements include:

- Even though **FFTMULT** accepts as parameters *real-valued* vectors, it creates internal working copies as *complex-valued* vectors, with initially zeroed imaginary parts. This is simple and needs a minimum amount of code, but is highly inefficient. For real-valued arrays there’s a simple trick that allows performing FFTs on them using complex vectors *half* the size, thus *half* the memory and running time.
- **FFTMULT** uses 4-digit groups for each vector element. This is so that you can multiply very large numbers without risking errors due to **FOUR** being unable to generate values which correctly round to the exact integers, due to *oversized* elements before releasing carries. For smaller sizes, it might be possible to use 5-digit groups for greater speed and less memory needed, while for larger sizes it might be necessary to stick to 3-digit groups. Using 4-digit groups is a decent compromise which seems to work flawlessly up to 1024-digit numbers (2048-digit results) at least.
- To *empirically* determine if the number of digits per group is adequate or else rounding errors might be creeping in, simply monitor if the values getting rounded (with **IROUND**) are *sufficiently near an integer*. Keeping track of the maximum difference (in absolute value) between each value and the nearest integer will warn you if the results are no longer reliable, say MDIF > 3/8.

Well, I sincerely hope you found the topic interesting. That one can multiply numbers this way is nothing sort of *miraculous* the first time you see it, and that it happens to be faster than the obvious way, despite the complex exponentials, approximate floating-point values and all, is *downright miraculous*, period. The greater miracle, however, is that it can be implemented so easily in the HP-71B. Agreed, an HP-49G++S-Whatever can multiply two reasonably large integers much faster than a 71B running this algorithm. But let the numbers get bigger and bigger, and *invariably* you’ll reach a point where the 71B will *win*. That is, unless you try and adapt this algorithm to run in your machine... Well ? Why don’t you give it a try ?