ELECTRONOTES
1016 Hanshaw Road
Ithaca, NY 14850

APPLICATION NOTE NO. 387

October 10, 2012

FUN WITH FFT TIMING

What is the difference between the DFT (Discrete Fourier Transform) and the FFT (Fast Fourier Transform)? The stock answer is that the FFT is a fast algorithm for calculating the DFT. This is almost true. Here are some facts:

The DFT of a length-N sequence x(n) is calculated by:

$$X(k) = \sum_{n=0}^{N-1} x(n)e^{-j(2\pi/N)nk} \qquad \text{for } k=0:N-1 \qquad (1a)$$

and the inverse DFT is:

$$x(n) = (1/N) \sum_{k=0}^{N-1} X(k)e^{j(2\pi/N)nk} \qquad \text{for } n=0:N-1 \qquad (1b)$$

We can also write this in matrix form:

$$\underline{\mathbf{X}} = D\,\underline{\mathbf{x}} \qquad (2a)$$

where D is an NxN matrix with $D_{p,q} = e^{-j(2\pi/N)pq}$ and the inverse is:

$$\underline{\mathbf{x}} = (1/N)\,DI\,\underline{\mathbf{X}} \qquad (2b)$$

where $DI_{p,q} = e^{\,j(2\pi/N)pq}$. Either of these forms, the summation equations (1a and 1b) or the matrix equations (2a and 2b) show that the direct calculation of the DFT (and/or inverse DFT) involves order N^2 operations.

The FFT is very well described in many many places. One or our previous Application Notes (AN-312 from 1990) gives an intuitive notion of how and why an FFT can be implemented. The FFT is NOT the same things as the DFT. One is a direct brute force calculation (the DFT) and the other is a very fast algorithm that is often, but not always, available (the FFT). While the DFT and the FFT are not the same thing, it is true that the FFT of a sequence x(n) and the DFT of the same sequence x(n) are exactly the same thing.

Because the FFT is not only generally faster, but much faster (the difference between practicality and impracticality in many situations), we generally want an FFT. Because computationally we usually use the FFT, we often use the terms "FFT" and "DFT" to mean the same thing. It is as though the term DFT becomes obsolete – perhaps so For example, when we want a DFT in Matlab, we use the ***fft*** function.

Originally, we thought of the FFT as an algorithm that could be applied when we had N that was a power of 2, such as 8, 256, or 2048, etc. Indeed we sometimes still see relatively new discussions saying that the FFT should be (must be!) a power of 2. This is NOT true. For one thing, we have a length-12 example in AN-312. For another, if we use Matlab, we can use any length sequence we want (16, 100, 77,. etc.) and ***fft*** gives us an answer. In fact, a complete presentation of the FFT ideas involves a length that can be factored. Further, it works best when we have a highly composite number. So clearly, a power of 2 is as good as it gets. But something like N=12 = 2x2x3 also shows an advantage for the FFT. Even something like N= 77 = 7x11 has a usable FFT. However, something like N=73 (prime) has no FFT. If you ask Matlab to compute the FFT of a length 73 signal, it apparently does the DFT. Because of the high speed of today's computers, perhaps the advantage of an FFT over a DFT is not apparent, especially if we are only computing a few. In fact, if you compute a length-1373 (prime) and a length-1376 (=2x2x2x2x2x43), both are essentially instantaneous. If you compute 1000 of them in sequence, you will likely be able to tell the difference, and to time it to some degree. In fact, Matlab provides functions ***tic*** and ***toc*** to do timing.

For example, lets compute the time it takes to compute 1000 length-1376 ffts. It is perfectly good to make all 1000 sequences the same and consisting of all ones. (There is a slight difference (longer time) if we use, for example, a length-1376 ***rand*** signal.)

```
tic;  for k=1:1000;  fft(ones(1,1376));  end;  toc

elapsed_time = 1.3650 sec
```

so each length 1376 FFT takes about 1.365 millisecond – the blink of an eye.   If we change the length to 1373 (prime) we get:

```
tic;  for k=1:1000;  fft(ones(1,1373));  end;  toc

elapsed_time =   33.2460 sec
```

That's 33+ seconds (easily measurable) so these take about 33.246 millisecond, again a blink of an eye <u>for a single one</u>. This second sequence is ever so slightly shorter, but takes much longer – the difference being whether or not significant factoring is possible. How would a power of 2 compare. Let's try the longer length-2048 case.

```
tic;  for k=1:1000;  fft(rand(1,2048));  end;  toc

elapsed_time =  0.2780 sec
```

Thus, while much longer, these run much faster, 0.278 milliseconds.   It should be clear that you can write a program that runs and records times. You might want to actually use more than 1000 sequences for more precise timing measurements.

Moving forward here, instead of ***tic*** and ***toc*** with many runs we will use a function ***flops*** (floating point operations) that is available in some (older) versions of Matlab. The function flops(0) resets the counter to zero and then flops reports the number of operations that subsequently were required.   The program ***ffttime*** computes the flops count for FFTs for length 1 to 270, and plots the results three different ways.

The full plot is shown in Fig. 1 here. This pretty much tells the whole story. Fig. 2 and 3 provide "zooms" for better detail.   Note the very low flops counts for the lengths that are powers of 2, and the quadratic trend for the prime number lengths. These prime length cases presumably are technically DFTs and not FFTs. Further we see that there are additional quadratic trends for the case where the length is 2xprime, 3xprime, 4xprime, and so on. Within this cluster of quadratic trends, we also see the lower flops counts for the lengths that are more composite (more factors). The quadratic trend for 2xprime is about half that of the primes. This means that the FFT is decomposed into a prime number of half the length and a whole bunch of length 2 DFTs (N/2 of them) and these cost us very little (a length 2 DFT just takes a sum and difference).

Fig. 2 shows a zoom-in for lengths of 98-132, and the descriptions given for Fig. 1 above are easier to see in this detail. The lengths are shown in terms of their most factored versions. This makes it easier to appreciate how the factoring reduces the number of operations, and thus increases the speed.

PROGRAM CODE ***ffttime.m***

```
% ffttime.m
clear all
clf
for N=1:270
   x=ones(1,N);
   flops(0)      % reset flops to 0
   fft(x);
   t(N)=flops;   % store the time
end
figure(1)
plot([1:270],t,'or')
hold on
plot([-10 280],[0 0],'k:')
axis([-10 280 -50000 600000])

figure(2)
plot([1:270],t,'or')
hold on
plot([-10 280],[0 0],'k:')
axis([97.5 132.5 -30000 170000])

figure(3)
plot([1:270],t,'or')
hold on
plot([-10 280],[0 0],'k:')
N=1:270;
NLN=3.3*N.*log2(N);
plot(N,NLN,'b')
axis([-10 280 -1000 8000])
```

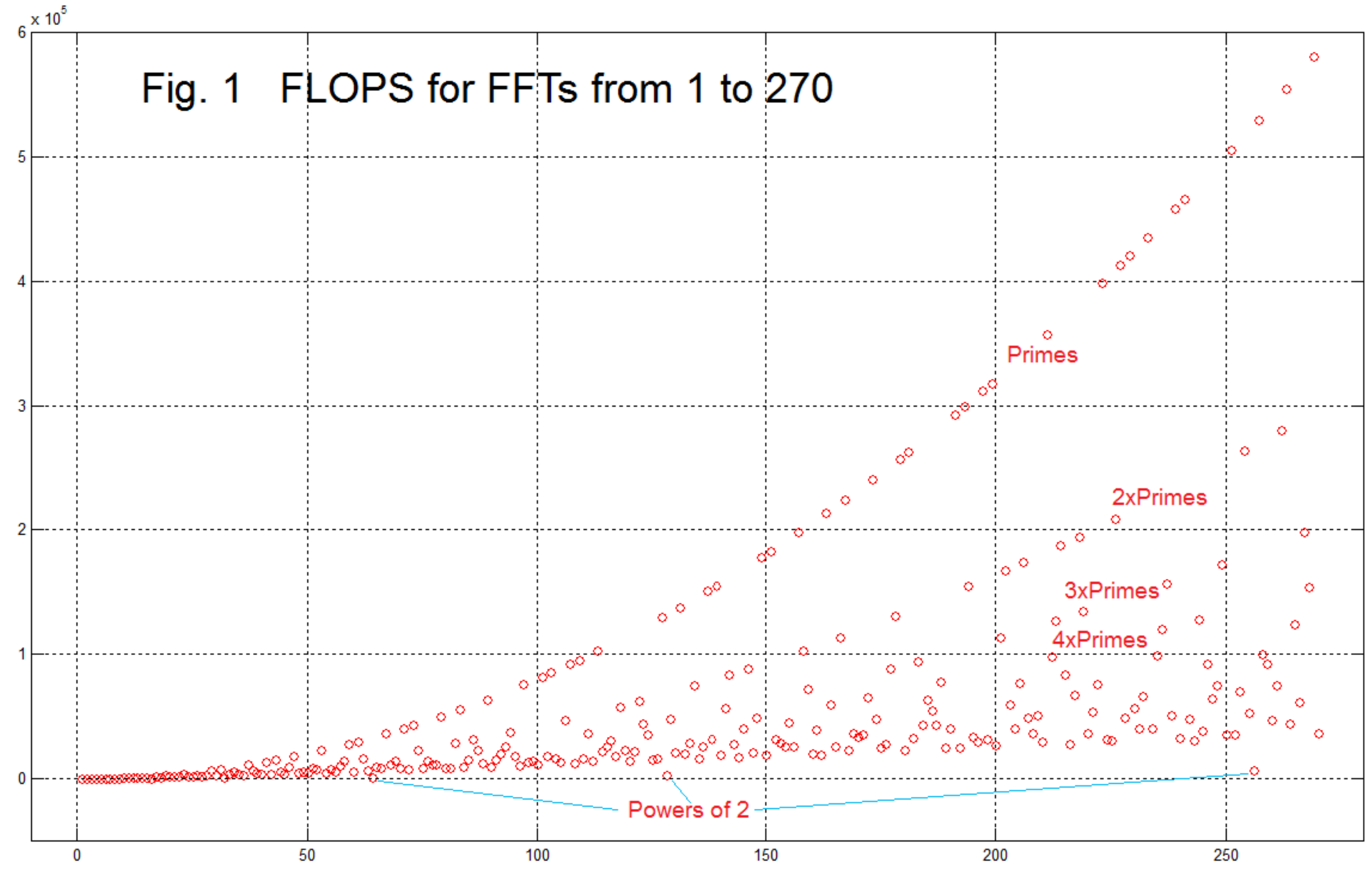

Fig. 1 FLOPS for FFTs from 1 to 270

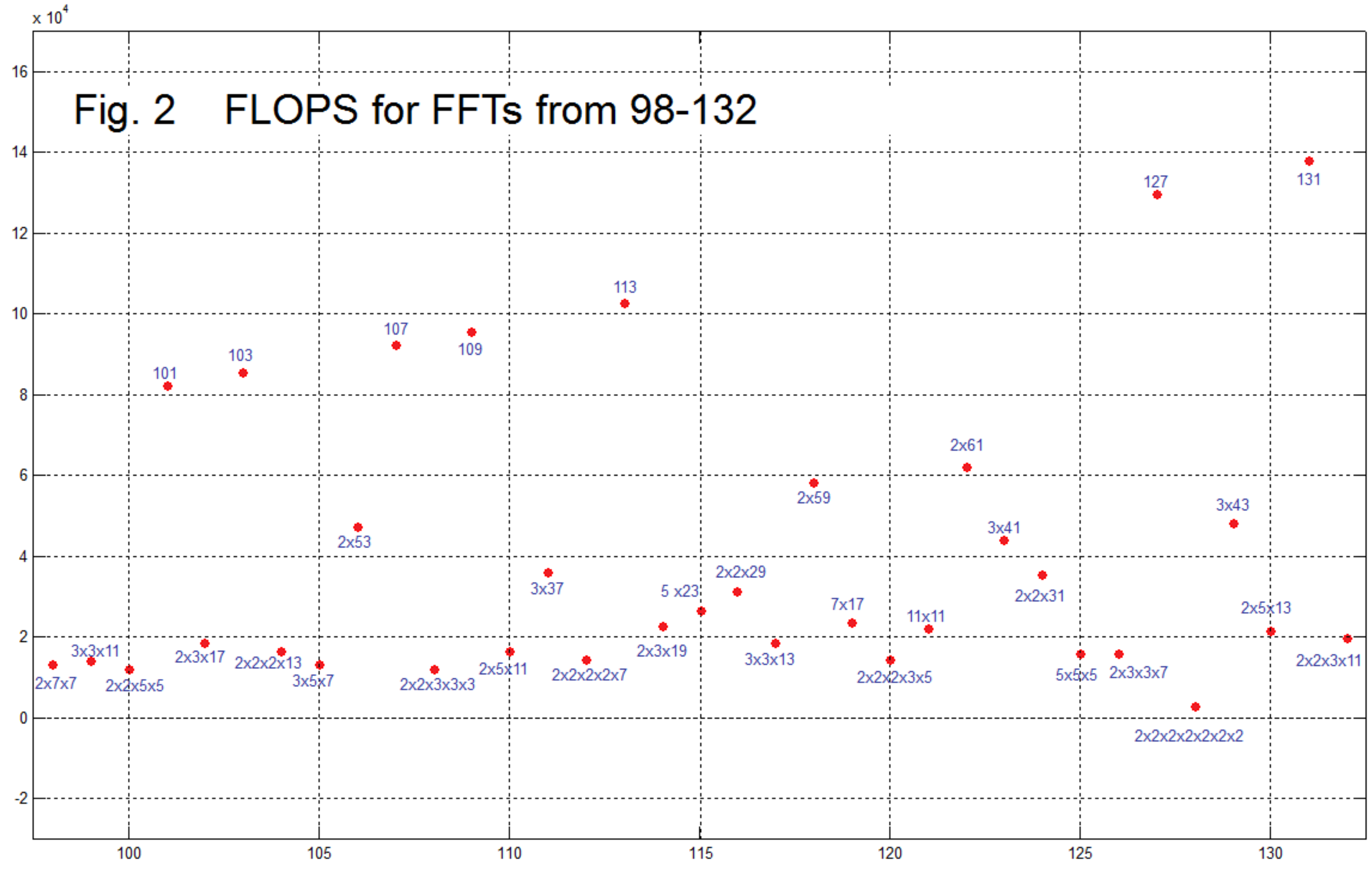

Fig. 2 FLOPS for FFTs from 98-132

Fig. 3 is a zoom for the low flops portion, intended just to show that the powers of 2 really are very fast. Many of us have heard (and studied) the notion that the operations required are of order $N\text{Log}_2(N)$, so we have plotted this curve, scaled experimentally (by multiplying by 3.3) to approximately fit the data. This is a reasonable fit. Perhaps the discrepancy seen is due to the time required to compute (set up) the test sequence, or any number of overhead (bookkeeping) issues such as reordering. The main point is clear.

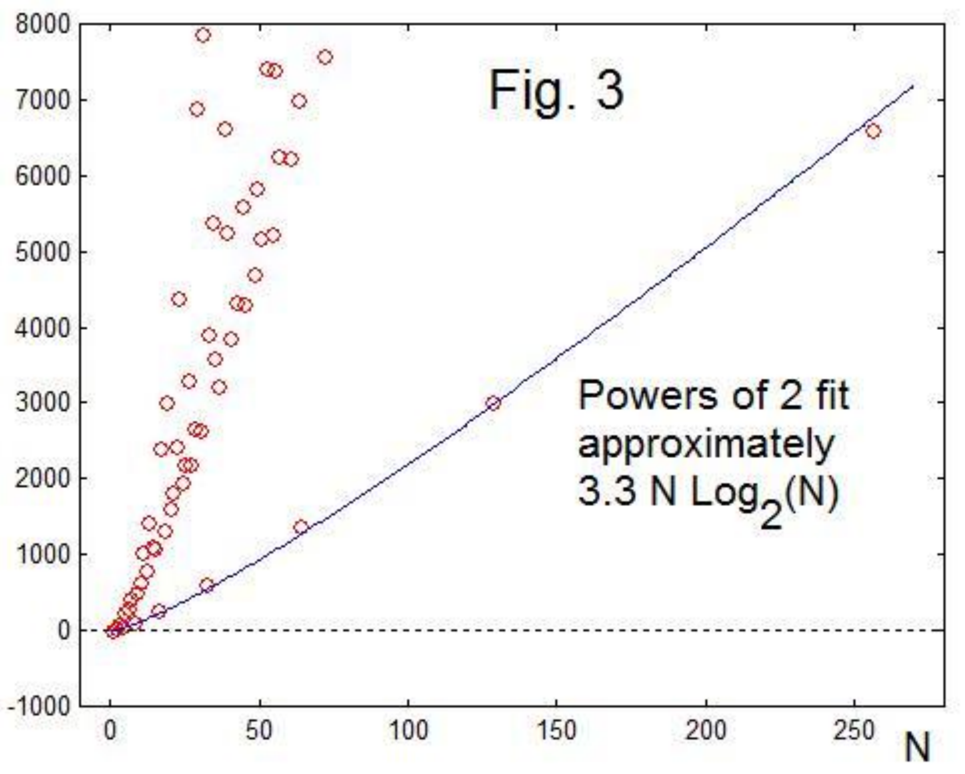
Fig. 3
Powers of 2 fit approximately 3.3 N Log2(N)
N