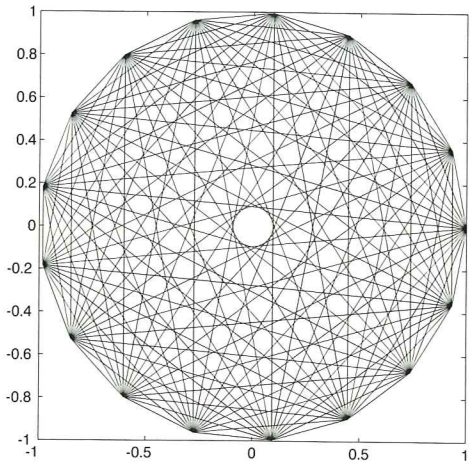
Yet Another Look at the FFT

Plots show structure of finite fourier transform

Try this:

```
plot(fft(eye(17)))
```

You will be drawing a picture of the Finite Fourier Transform of the identity matrix of order 17. You should see:



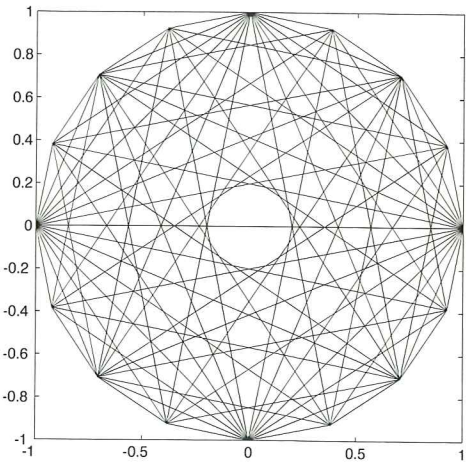
(You might want to set `axis('square')` and `axis([-1 1 -1 1])` first, and you'll see line types and colors we're not reproducing here.)

There are 17 points around the unit circle in the complex plane. Each point is connected to every other point. This is the complete graph on 17 points.

But now try 16 points:

```
plot(fft(eye(16)))
```

You should see:



There are 16 points around the circle, but some of the connections are missing. There is a horizontal line, but no matching vertical line. In fact, some of the points are connected to only 8, 12, or 14 others, rather than the 15 others that would be necessary to get a complete graph.

What's going on here? Well, it has something to do with the fact that 17 is a prime number, but 16 is a power of 2. Why do we care? Because a complete explanation sheds some light on the behavior of the `fft` algorithm. For 16 points there is a *Fast* Finite Fourier Transform (that's `ffft` with three `f`s), but for 17 points there isn't. The missing lines in our graph are directly related to the speed of the Fourier transform algorithm.

We're not talking minor speed differences. On the ancient 10 MHz PC laptop where I happen to be working, the MATLAB `ffft` on 512 points takes less than half a second, while the `fft` on 511 points over 7 seconds. (And, the `fft` on 513 points takes about 2.3 seconds because 513 = 3*3*3*19, but I'm getting ahead of myself.)

Chuck Denham first showed me these `plot(fft(eye(n)))` pictures a few years ago, but I didn't really understand them until I decided to write about them for the newsletter.

While investigating these `fft` plots, I learned a lot from a new book published by SIAM, *Matrix Frameworks for the Fast Fourier Transform*, by Charles Van Loan. Charlie is a big MATLAB fan. His new book uses MATLAB notation. Another book he wrote with Tom Coleman, *Handbook for Matrix Computations*, also published by SIAM, has a chapter on MATLAB. But I don't think he knows about `fft` plots; I hope he reads this newsletter.

Here is a partial explanation. Let `x` be a column vector of length `n`. The `fft(x)` could be computed by a matrix-vector product

```
fft(x) = F*x
```

where `F` is the complex `n`-by-`n` Fourier matrix whose elements are powers of ω, an `n`th root of unity:

```
j = sqrt(-1)
omega = exp(-2*pi*j/n)
for i=0:n-1, for k=0:n-1
   F(i+1,k+1) = omega ^ (i*k);
end, end
```

Van Loan calls `F` the `DFT` matrix, for Discrete Fourier Transform. It is within a scale factor of being a unitary matrix:

```
F * F' = n*I
```

where `I = eye(n,n)`. So the inverse of `F` is `F'/n`. This is why the inverse `fft` is just like the `fft` itself, with a sign flip and a scale factor of `n`.

Using `F*x` to compute `fft(x)` for a vector of length `n`

would require n^2 multiplications. If we double the length of `x`, it would take four times as long to compute `fft(x)`. This is essentially what happens for some values of `n`, like `n = 511`. But for other values of `n` the computation can be done with many fewer operations.

The key to fast algorithms is to find a "sparse factorization", `F = F1*F2*...*Fm` where each `Fk` has only a few nonzero elements. Then `F*x` can be computed with fewer operations.

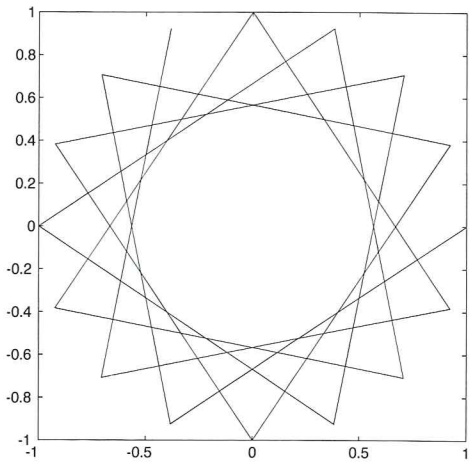
If `n` is an even number, it is always possible to express `F` as the product of two sparse matrices, `F = F1*F2`. Repeating this factorization, if `n` is a power of 2, say $n = 2^m$ where `m = log2(n)`, then it is possible to express `F` as the product of `m` matrices, each of which has only four nonzero elements, and a permutation. This leads to an algorithm involving only `4*n*log2(n)` arithmetic operations and some fancy subscripting.

Let `I = eye(n)`. The plot of `fft(I)` is the same as `plot(F)`. It is the union of the `n` pictures, `plot(fft(I(:,k)))`. But `fft(I(:,k)) = F(:,k)` is a vector of powers of ω, which gives the points on the unit circle in the complex plane. To see the individual plots, we can use the general form of a MATLAB `for` statement where the loop variable is successively assigned the columns of a matrix.

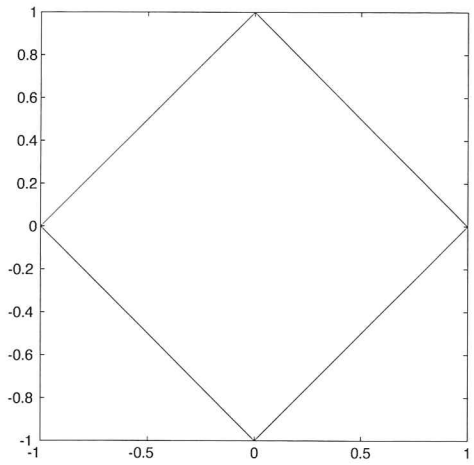
```
for f = fft(eye(n))
    plot(f)
    pause
end
```

Here is `plot(F(:,6))` when `n = 16`. You can see the line start at ω^5, go to ω^{10}, then ω^{15}. The next point would be ω^{20} but, because $\omega^{16} = 1$, this is the same as ω^4. Then comes ω^9, ω^{14}, $\omega^{19} = \omega^3$, and so on. The exponents on the powers of ω are obtained by taking the multiples of 5 and reducing them modulo `n = 16`. In this case, we eventually touch all 16 points.

But here is the `plot F(:,5)` for `n = 16`. We start at ω^4,



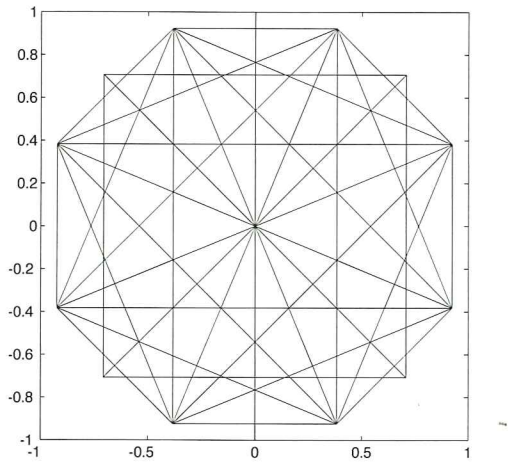
go to ω^8, then ω^{12}, then $\omega^{16} = 1$, then back to ω^4. We see only four line segments connecting the four points, because each segment has actually been drawn four times.



In general, the `plot(fft(I(:,k+1)))` starts at ω^k and connects to the powers of ω with exponents `rem(2*k,n)`, `rem(3*k,n)`, `rem(4*k,n)`, etc. Now here is the key point. If `n` is prime, then `rem(j*k,n)` will not be zero until `j` or `k` is equal to `n`, so none of the powers of ω are repeated and we get the complete graph on `n` points. But if `n` is not prime, then `rem(j*k,n)` will be `0` when, for example, `j*k = n`, so some powers of ω will be repeated and others will be skipped.

As Van Loan (and hundreds of other writers) explain, the possibility of getting a Fast Finite Fourier Transform by factoring `F` into `F1*F2*...*Fm` with sparse `Fk` is directly related to the possibility of factoring `n` into a product of powers of small primes. And, as we have seen here, this is directly related to structure of the graph obtained by `plot(fft(eye(n)))`.

I'm not completely satisfied with the explanation I've presented here. It is not very precise. In particular, exactly what line segments are *missing* from `plot(fft(eye(n)))`? Here is a plot of the missing segments when `n = 16`; this graph is the complement of the second graph shown earlier.



I won't tell you how I generated this plot because it's not very elegant. If anybody can tighten up this discussion and give a clean, number-theoretic characterization of these graphs and their relationship to the operation counts for `fft`, I'd sure like to hear about it.