Top Ten Algorithms Class 8

John Burkardt
Department of Scientific Computing
Florida State University

.....

http://people.sc.fsu.edu/∼jburkardt/classes/tta_2015/class08.pdf

19 October 2015

Our Current Algorithm List

1. Bank routing number checksum for error detection
2. Bernoulli number calculation
3. Bootstrap algorithm
4. Data stream: most common item
5. Euclid's greatest common factor algorithm
6. Hamming error correcting codes
7. ISBN (International Standard Book Number) checksum
8. k-means clustering algorithm
9. Luhn/IBM checksum for error detection
10. Monte Carlo Sampling

Our Current Algorithm List

1. PageRank algorithm for ranking web pages
2. Pancake flipping algorithm for genome relations
3. Path counting with the adjacency matrix
4. Power method for eigenvector problems
5. Probability evolution with the transition matrix
6. Prototein model of protein folding
7. QR (Quick Response) images and error correction
8. Reed-Solomon error correcting codes
9. Search engine indexing
10. Trees for computational biology
11. UPC (Universal Product Code) checksum for error detection

Student Presentation

AmirHessam Tahmassebi, "Discrete Cosine Transformations - How JPGs work"

Reference: Nick Berry, http://datagenetics.com/blog/november32012/index.html

A simple format for a color image that is MxN pixels would store R, G and B values for each pixel (3 * 8 bits * M * N).

The JPEG format tries to compresses visual information without greatly disturbing the quality of the image. To do this, it relies on a version of the Fast Fourier Transform known as the Discrete Cosine Transform.

Philosophy

We think of a function as a graph, which we feel could be a completely arbitrary, "very infinite" list of pairs of values (x, y).

For computational science, many things we work with are really functions, but to put them on the computer, we have to discretize them. That starts with the number system itself (remember machine epsilon). But it continues with functions and images and sound recordings and videos.

We try to represent infinite continuously varying quantities by finite things that jump.

It really helps if we can discover that the infinite, continuous thing has a natural decomposition, a good approximation in our finite discrete world.

You are probably used to one way to discretize functions, which is to represent them as a polynomial. The powers of x become our "building blocks".

Philosophy

The first simplification we use is **continuity**, which says that the value of $f(x_0)$ actually gives us some information about values of $f(x)$ at nearby x's.

Today we analyze objects for which another feature can be exploited. A periodic function has the property that the sequence of values repeats after a certain time.

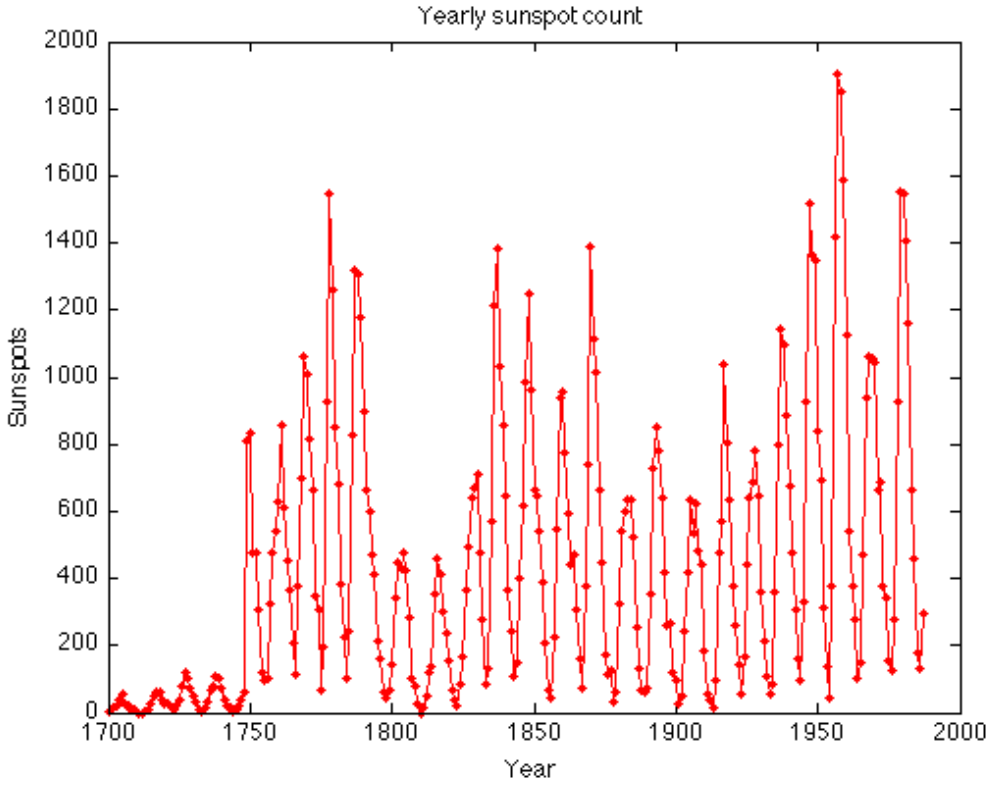
Examples include music, images, records of sunspot activity, snowfall data, stockmarket prices, the number of animal species.

In such cases, the value of $f(x_0)$ tells us something about the value of $f(x)$ at nearby values of x, but also for values at $x_0 + p$ and $f(x_0 + 2p)$ and so on.

In mathematics, a periodic function repeats exactly and repeats forever. We will want to consider more general cases where the repetition might be approximate, and might only extend a certain number of cycles.

What's the Period, Kenneth?

The most basic fact about a periodic function is the value of the period itself. For snowfall data, the period is a year. For sunspots, it took many years of observation to notice a period of 11 years. Notice that we are not talking about exact repetition, but some kind of natural rhythm in the data.



Estimate the Period

Suppose we can't see a pattern in the data visually. For simplicity, assume that our data is equally spaced, and is sampled over exactly some number of periods. Can we get our data to tell us what that period is?

If we treat our discrete data as a vector $y()$, then we can compute the dot product

$$\text{yty}(0) = y(1:n)'y(1:n) = y(1)*y(1)+y(2)*y(2)+...+y(n)*y(n).$$

Imagine the data on an x axis that wraps around, with the y data forming spikes, so our data has made a sort of crown. To symbolize our dot product, we can nest a second crown inside the first, with the spikes matching up.

What would happen if we turned the inner crown one position?

Estimate the Period

Turning the crown one position is carrying out a circular shift.

$$\text{yty}(1) = y(1:n)'y(2:n{+}1) = y(1){*}y(2){+}y(2){*}y(3){+}...{+}y(n){*}y(1)$$

Surprisingly, this result should be **less** than what we got before. It's more or less true for the same reason as $xy + xy <= x^2 + y^2$ *(Can you prove this?)*

We can turn the data another notch, computing $yty(2)$, and so on, and we should never get a value as big as the first one.

However, if the data is mathematically periodic, we will exactly compute the same value as $yty(0)$ before we have made a full turn. And if the data at least has a periodic component, then every now and then, the data will approximately match.

Maximums in the value of *yty* suggest when periodicity occurs.

Estimate the Period

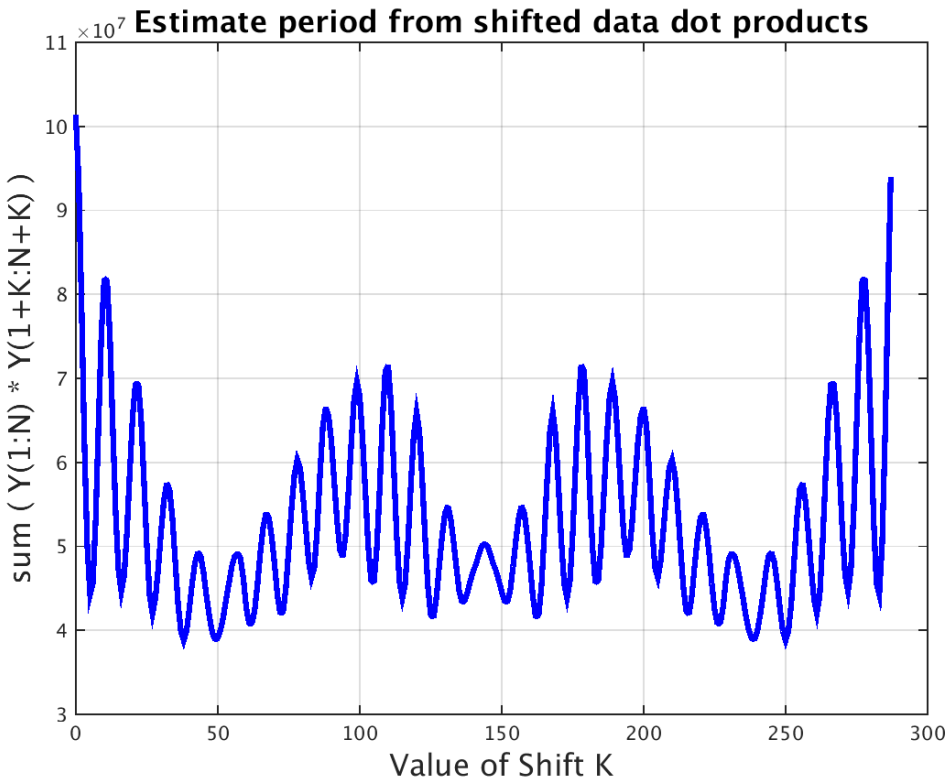
Look for local maximums in the data when we compute $\sum_{k=0}^{n-1} y(1:n) * y(1+k:n+k)$

0	1.01022e+08	11	8.16740e+07	22	6.93776e+07
1	9.36642e+07	12	7.49436e+07	23	6.51396e+07
2	7.88188e+07	13	6.43848e+07	...	
3	6.26464e+07	14	5.42281e+07		
4	5.02599e+07	15	4.72126e+07		
5	4.42861e+07	16	4.45973e+07		
6	4.56649e+07	17	4.6447e+07		
7	5.32449e+07	18	5.16854e+07		
8	6.45323e+07	19	5.88706e+07		
9	7.54103e+07	20	6.54832e+07		
10	8.18572e+07	21	6.93683e+07		
11	8.16740e+07	22	6.93776e+07		

(To be fair, year 10 is slightly bigger than year 11!)

Estimate the Period

The *yty* plot shows 11 and perhasp 50 year periodicity.

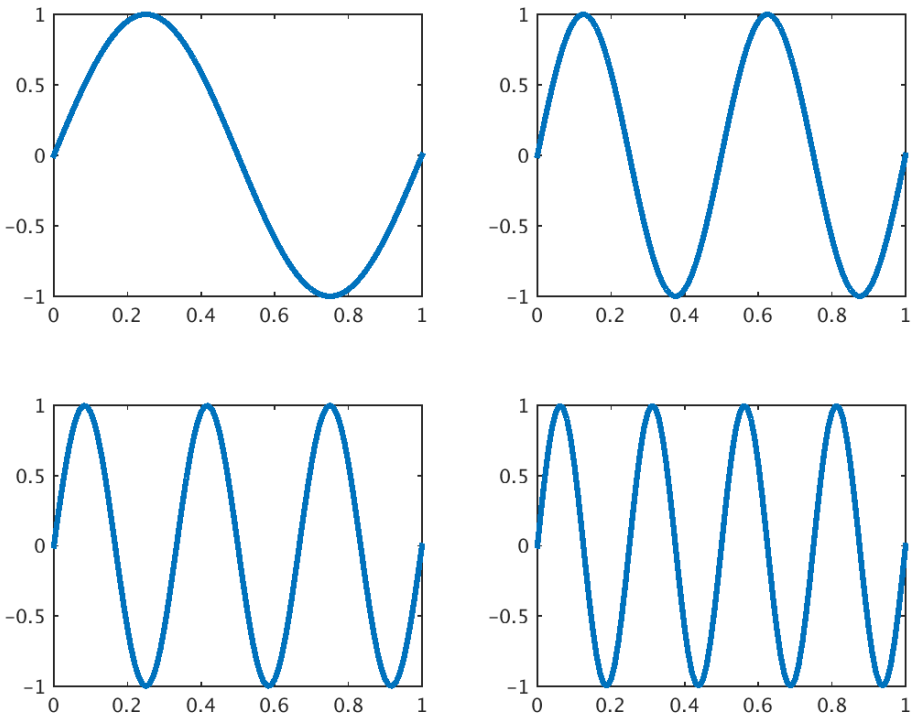


The *yty* is a *discrete autocorrelation function*; this process is the discrete autocorrelation period estimation algorithm.

Using a Harmonic Basis

Where do periodic functions come from? We can build periodic functions with a harmonic basis of sine functions. We choose a fundamental period P, and construct $\phi_k(x) = \sin(2k\pi x/P)$ for $k = 1, ..., kmax$.

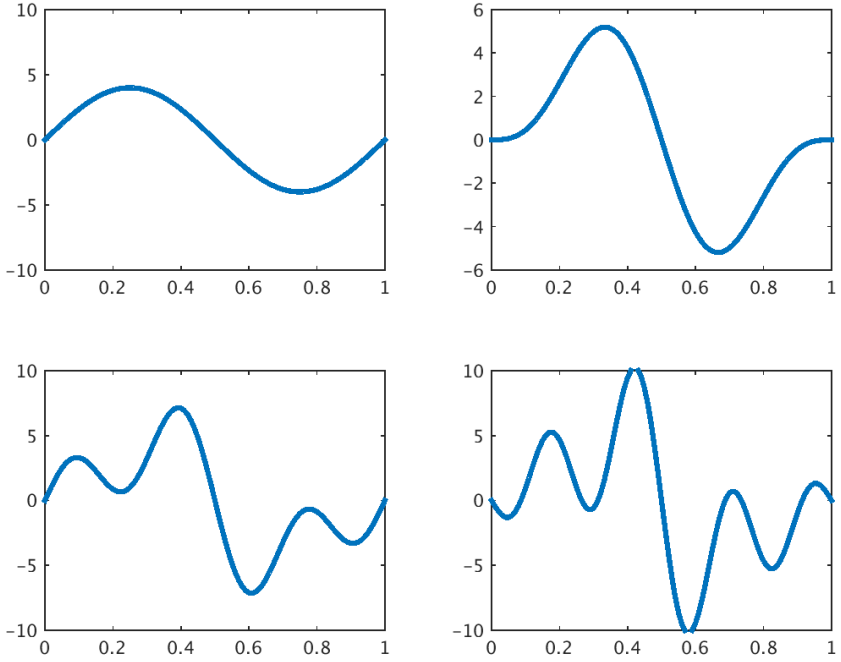
Function $\phi_k(x)$ will wiggle exactly k times over the period P, and thus has frequency k/P.



Combining Harmonic Basis Functions

Now we can build a function with period P:

$$f(x) = \sum_{k=1}^{kmax} c_k \sin(2k\pi x/P)$$



Here, we build up $f(x) = 4\phi_1(x) - 2\phi_2(x) + 3\phi_3(x) - 4.0\phi_4(x)$

Combining Harmonic Basis Functions

```
  n = 501;
  p = 10;
  x_min = 5.0;
  x_max = 5.0 + p;
  x = linspace ( x_min, x_max, n );
  phi1 = sin ( 1 * ( 2 * pi / p ) * x );
  phi2 = sin ( 2 * ( 2 * pi / p ) * x );
  phi3 = sin ( 3 * ( 2 * pi / p ) * x );
  phi4 = sin ( 4 * ( 2 * pi / p ) * x );
  c1 =  4.0;
  c2 = -2.0;
  c3 =  3.0;
  c4 = -4.0;
  f = c1 * phi1 + c2 * ph2 + c3 * phi3 + c4 * phi4;
  plot ( x, f, 'r-' )
  title ( 'F=4\phi_1(x)-2\phi_2(x)+3\phi_3(x)-4\phi_4(x)' );
}
```

Listing 1: Compose Periodic Function

Detecting Harmonic Basis Functions

Suppose we see the function $f(x)$...or, equivalently, someone sends us an electronic signal, or sings a song, or shows us a graph. Assuming we know the period P, can we determine the coefficients c_1, c_2, c_3, c_4 used to create $f(x)$ from the basis functions?

Since we are computational, let's assume that F is a set of n equally-spaced samples of $f(x)$ over the period P. To do the analysis, sample each basis function at the same points; Φ_1 will contain n values of $\phi_1(x)$ over the period.

To discover the coefficients, we just need dot products:

$$
\begin{aligned}
c1 =& F'\Phi_1/\Phi_1'\Phi_1 \\
c2 =& F'\Phi_2/\Phi_2'\Phi_2 \\
c3 =& F'\Phi_3/\Phi_3'\Phi_3 \\
c4 =& F'\Phi_4/\Phi_4'\Phi_4
\end{aligned}
$$

Decompose a Periodic Function

```
  n = 501;
  p = 10;
  x_min = 5.0;
  x_max = 5.0 + p;
  x = linspace ( x_min, x_max, n );
  x = x';
  phi1 = sin ( 1 * ( 2 * pi / p ) * x );
  phi2 = sin ( 2 * ( 2 * pi / p ) * x );
  phi3 = sin ( 3 * ( 2 * pi / p ) * x );
  phi4 = sin ( 4 * ( 2 * pi / p ) * x );
  F = f ( x );

  c1 = ( F' * phi1 ) / ( phi1' * phi1 );
  c2 = ( F' * phi2 ) / ( phi2' * phi2 );
  c3 = ( F' * phi3 ) / ( phi3' * phi3 );
  c4 = ( F' * phi4 ) / ( phi4' * phi4 );

  f2 = c1 * phi1 + c2 * ph2 + c3 * phi3 + c4 * phi4;
  plot ( x, f2, 'r-' )
  title ( 'Recovered function' );
}
```

Listing 2: Decompose Periodic Function

Remarks

The computation of c_4 is independent of the computation of other coefficient. We can compute any coefficient we want at any time.

If $f(x)$ gets corrupted by (relatively small) random noises, this will change all the results, but we will still be able to recover the coefficients (approximately).

As long as the function $f(x)$ is real, and we are satisfied with the boundary conditions, we can use only sines or only cosines. The general analysis involves complex data functions and coefficients, and both sine and cosine basis functions.

If a formula for the function $f(x)$ was available, we could use integrals instead of dot products:

$$c_k = \int_a^b f(x)\sin(k*2\pi x/P)dx / \int_a^b \sin(k*2\pi x/P)\sin(k*2\pi x/P)dx$$

The Discrete Fourier Transform

The harmonic sine example is a special case. In order to handle the general case, we need to use the discrete Fourier transform, which we can symblize by $F()$. If we start with an N-vector g of real data, then there are two steps in this process:

- forward transform: $G = F(g)$, data to complex Fourier coefficients.
- inverse transform: $g = F^{-1}(G)$, complex Fourier coefficients to data.

When we use the forward transform, we are analyzing the data. The values in G are the strengths of the signal in various frequencies. If $G(i)$ is very large, then we have detected a periodic behavior of the corresponding frequency. If a signal originally was composed of only a few frequencies, but then had noise added, we will see a few large values in G, and many small ones. If we zero out the small G's, then the inverse transform will give us the approximate original signal.

The "Slow" Fourier Transform

We can write the forward Fourier Transform as a matrix multiplication:

$$G = F * g$$

where

$$\theta_{i,j} = 2\pi i\, j/n$$

and

$$F_{i,j} = \cos(\theta_{i,j}) - I\ \sin(\theta_{i,j})$$

and the inverse transform F^{-1} is simply

$$F_{i,j}^{-1} = \cos(\theta_{i,j}) + I\ \sin(\theta_{i,j})$$

The Slow Fourier Transform (Forward)

```
function y = c8vec_sftf ( n, x )

  y(1:n) = 0.0;

  for k = 1 : n
    for j = 1 : n
      theta = - 2.0 * pi * ( k - 1 ) * ( j - 1 ) / n;
      y(k) = y(k) + ( cos ( theta ) - i * sin ( theta ) ) *
          x(j);
    end
  end

  y(1:n) = y(1:n) / sqrt ( n );

  return
end
```

Listing 3: Slow Fourier Transform (Forward)

The Backward or Inverse Transform is computed by code that is identical, except that the sine term is positive.

From Slow To Fast Fourier Transform

It's easy to see from the **c8vec_sftf.m** code that the slow Fourier Transform has complexity $O(n^2)$. This means that we can probably use the algorithm for n=1,000 but processing data of size n=1,000,000 will take a million times longer. A computer capable of doing one multiply and add every microsecond would take about 11.5 days to process the data.

The Fast Fourier Transform was developed so that large data could be handled. The FFT has complexity $O(n \log(n)$. Our example of a vector of a million values could be done in 20 seconds rather than 11.5 days!

If you think it unlikely that you would want to process one million data values, consider that an image file can easily contain a million pixels. Image files can be processed by an FFT that first treats each column as a vector, then each row. If there are about 1,000 columns and rows, then the slow Fourier transform of an image would again take 11.5 days.

An FFT of N items is 2 FFT's of N/2 items

The key to the fast FFT is a sort of recursion. Recall that

$$e^{-2\pi i/n} = \cos(2\pi i/n) - i\sin(2\pi/n)$$

Let's write

$$\omega_n = e^{-2\pi i/n}$$

and notice that

$$\omega_{2n}^2 = \omega_n$$

Getting to N LogN

The Fourier Transform can be written

$$G_k = \sum_{j=0}^{n-1} \omega^{jk} * g_j$$

If n is even, we can split this into odd and even values of j:

$$G_k = \sum_{\text{even j}} \omega^{jk} * g_j + \sum_{\text{odd j}} \omega^{jk} * g_j$$

$$= \sum_{j=0}^{n/2-1} \omega^{2jk} * g_{2*j} + \omega^k * \sum_{j=0}^{n/2-1} \omega^{2jk} * g_{2*j+1}$$

Assuming $n/2$ is even, we can repeat this process, and if n is a power of 2, we eventually reach n FFT's of length 1. There is $O(n)$ work at each step, and $log(n)$ steps.

Thus, when you can choose N, the number of data values samples, researchers prefer a power of 2.

Getting to N LogN

If N cannot be chosen, then researchers will often pad the data with enough zeros to create an enlarged vector whose size is a power of 2. (The FFT of a padded vector will not be identical to the SFT of the unpadded data; however, we hope it will be close.)

If N is not a power of 2, but is factorizable, some efficiency can still be gained. An FFT of 125 values is 5 FFT's of 25 values is 25 FFT's of 5 values...at which point you can set up the 5x5 FFT matrix and multiply it by each of your subvectors.

Another issue arises if you have data that is not equally spaced. Perhaps the simplest way to proceed is to compute the interpolant to your data, and then sample it at equally spaced increments.

Also, it is common to "de-trend" data by subtracting the linear function that matches the first and last values, to make the periodic behavior easier to analyze.

Sampling and detrending issues will arise in the fossil example.

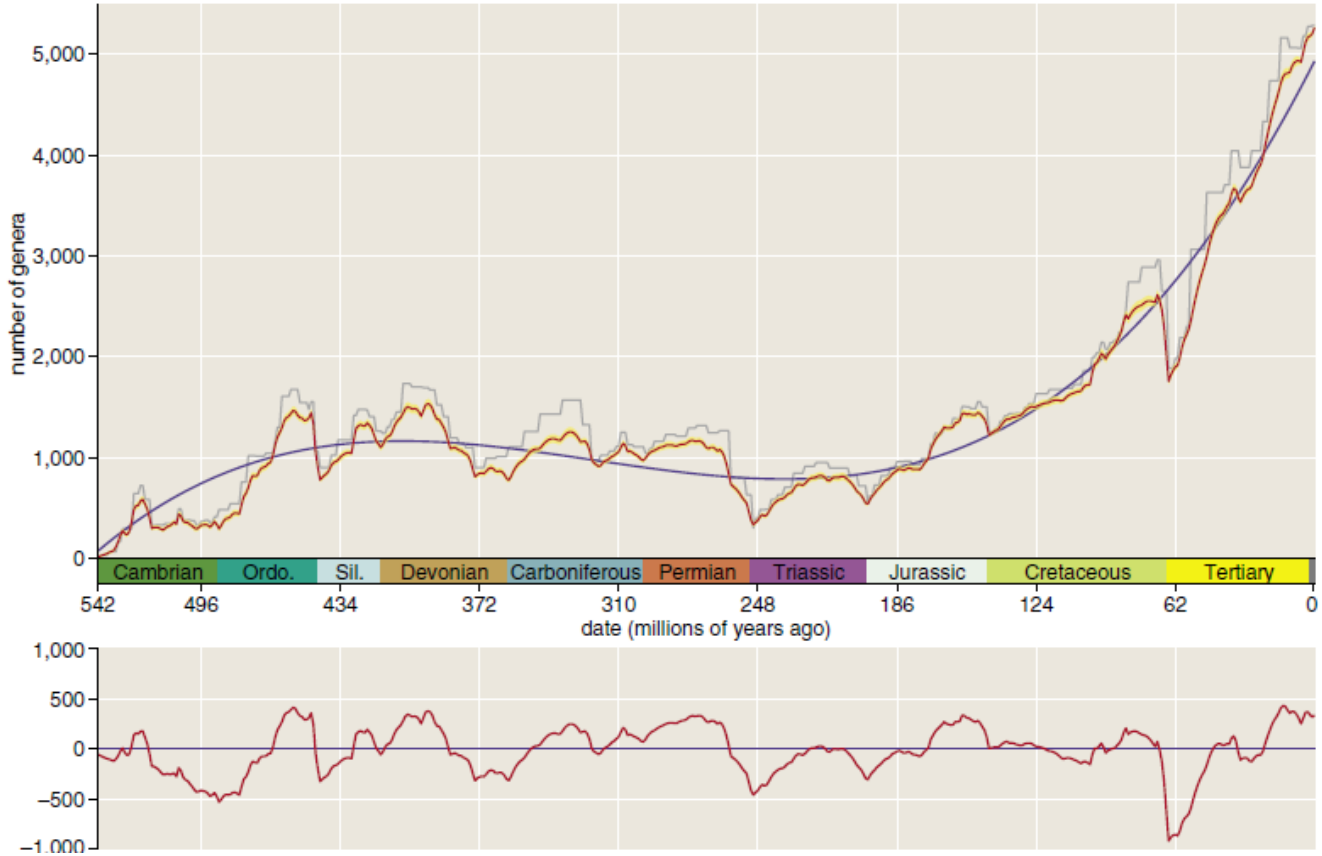
Periodic Signal in Species Data

In the article "Life Cycles", Brian Hayes asks whether there have been periodic increases and decreases in the number of living species. If this is so, we might want to follow up by looking for astronomical, geological, or biological explanation.

The data starts with a list of the earliest and lastest fossil evidence of a member of a given genus of creatures, for more than 36,000 such records, extending over a period of about 550,000,000 years. We can then divide the geological timeline into equally spaced units of roughly 3,000,000 years in extent and now we have equally spaced samples of a function.

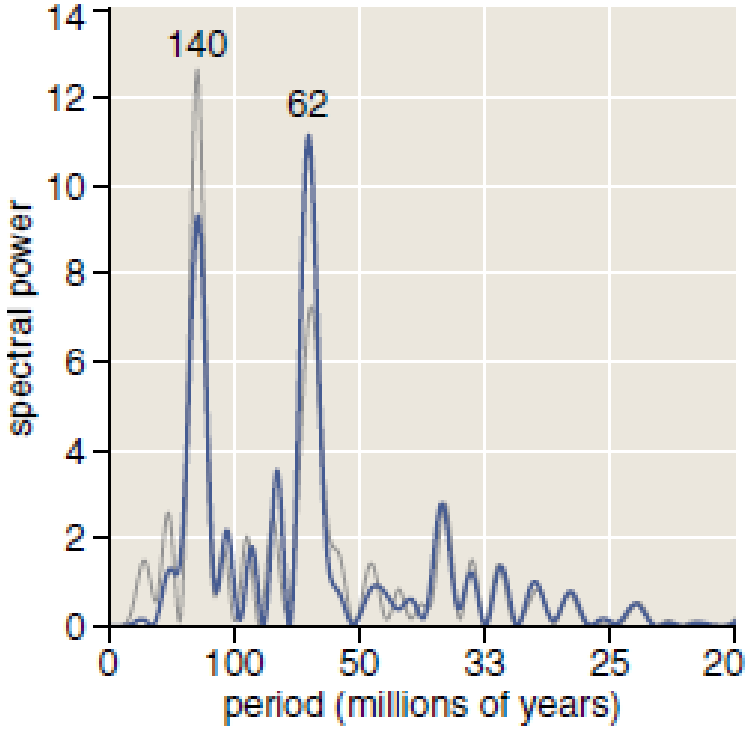
We know the function is not periodic; what we are looking for is a periodic component, that is a signal that behaves like some multiple of sine function at a given frequency.

Periodic Signal in Species Data



The top graph gives the raw data, as well as a cubic function fit to the data. The lower graph shows the raw data minus the cubic fit. This is the curve we will analyze with the FFT.

Periodic Signal in Species Data



The FFT coefficients are complex, and come in pairs $(a + bi)$. Each pair is associated with a frequency. The strength of the signal at this frequency is $\sqrt{a^2 + b^2}$, the "spectral power". We see strong signals at frequencies of 140 and 62 million years.

Conclusion

There are many variations on the Fourier transform, including versions that use only sine or cosine functions. When you know your input data is real, you can use a reduced version of the FFT that only computes half the coefficients, because the second half of the data is the mirror image of the first.

Related transforms include the Walsh, Haar, Hartley and wavelet transforms. Most of these have the property that they can be thought of as multiplication by a matrix (forward transform) or its complex conjugate transpose (inverse transform).

One of the best FFT packages available now is known as FFTW, the "Fastest Fourier Transform in the West" **http://www.fftw.org/**. The FFT used in MATLAB is now based on FFTW.

Student Volunteer?

Several telecom companies are fighting for a contract to maintain cellphone towers. One difficulty is that they must assign a frequency to each tower, in such a way that nearby towers don't use the same one.

I have discovered such a scheme, and I want to offer it to the highest bidder. Until I have all the bids, I don't want to reveal my scheme, otherwise any of the bidders can steal it. But the bidders don't want to bid unless they believe I really have a scheme that works.

Is there a way to convince each bidder that I've got a scheme that works, without revealing the actual scheme?

Reference: Matthew Green, *Zero Knowledge Proofs*, http://blog.cryptographyengineering.com/2014/11/zero-knowledge-proofs-illustrated-primer.html

Student Volunteer?

I am watching a radar screen that is updated every minute.

The radar screen is a 500x500 array of pixels which are OFF or ON.

The radar signal is noisy, so that each pixel has a chance of turning on even though there's nothing there.

I know there are two intruders somewhere on my screen. One is not moving, and one is moving at a constant speed and direction. After observing the screen for 30 minutes, can I detect the location of either intruder?

Reference: Nick Berry, *Detecting Targets in Noisy Radar Signals*, http://datagenetics.com/blog/may22014/index.html