FOURIER TRANSFORMS

Ari Maller

FOURIER SERIES

- Fourier transforms are a basic tool of physics and mathematics. It is also an important approach for certain numerical problems.
- Fourier transforms allow us to break down functions or signals into their component parts and analyze, smooth or filter then, and it gives us a way to rapidly perform certain types of calculations and solve certain differential equations.
- We have all learned that a periodic function f(x) defined between 0 and L can be expressed as a Fourier series

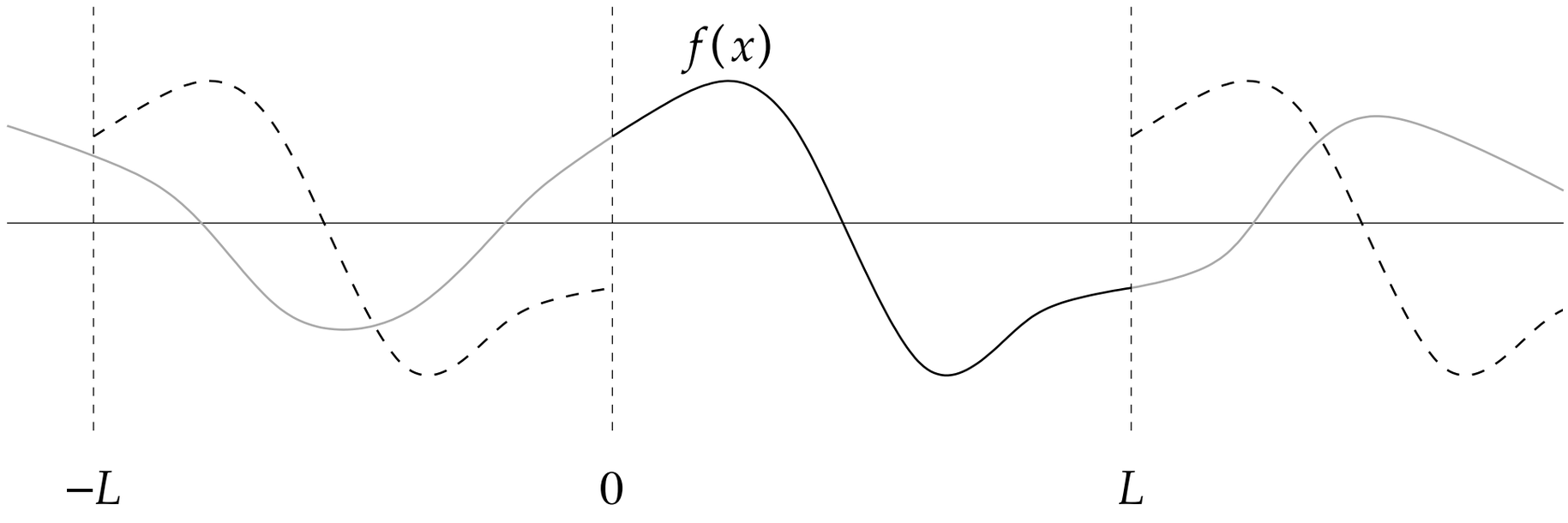
$$f(x) = \sum_{k=0}^{\infty} \alpha_k \cos\left(\frac{2\pi k x}{L}\right) \qquad \text{or} \qquad f(x) = \sum_{k=1}^{\infty} \beta_k \sin\left(\frac{2\pi k x}{L}\right)$$

FOURIER SERIES

- An alternative way to represent the general sin/cos series is

$$f(x) = \sum_{k=-\infty}^{\infty} \gamma_k \exp\left(i\frac{2\pi kx}{L}\right)$$

- Since the complex series includes both sine and cosine we will use it for most of our calculations.
- Note that the Fourier series can only be used for periodic functions. However, if we have a function that is not periodic we can simply replicate it over some range and then it will be.



- The coefficients γ_k are in general complex. The general way to calculate them is to evaluate the integral

$$\int_0^L f(x) \exp\left(-i\frac{2\pi kx}{L}\right) dx = \sum_{k=-\infty}^{\infty} \gamma_k \exp\left(i\frac{2\pi(k'-k)x}{L}\right) dx$$

- this is zero if k' ≠ k. So the only nonzero term in the sum gives

$$\gamma_k = \frac{1}{L}\int_0^L f(x) \exp\left(-i\frac{2\pi kx}{L}\right) dx$$

- Given the function f(x) we can find the Fourier coefficients γ_k by integration. We can also recover the function f(x) from the coefficients by performing the sum.

$$f(x) = \sum_{k=-\infty}^{\infty} \gamma_k \exp\left(i\frac{2\pi kx}{L}\right)$$

DISCRETE FOURIER TRANSFORM

- There are many situations where the pervious integral over f(x) can not be done analytically, either because of its functional form, or because there is no function but just a series of values from experiment or numerical calculations.
- In such cases we can evaluate the coefficients numerically, for example using the trapezoidal rule.

$$\gamma_k = \frac{1}{N}\sum_{n=0}^{N-1} f(x_n)\exp\left(-i\frac{2\pi k x_n}{L}\right)$$

- Here we have used the fact that f(L)=f(0) and defined x_n = nL/N. The points should be equally spaced for the trapezoidal rule. This form is very convenient for computation since we often know the value of a function over evenly spaced points.

DISCRETE FOURIER TRANSFORM

- This form is known as the discrete Fourier transform or DFT. The coefficients are defined slightly differently as

$$c_k = N\gamma_k = \sum_{n=0}^{N-1} f(x_n) \exp\left(-i\frac{2\pi k x_n}{L}\right)$$

- for no good reason, but it is the standard way. We will call c_k Fourier coefficients even though you need to divide them by N to get the real Fourier coefficients.
- The discrete Fourier transform is not an approximation, it is exact. That is the coefficients times the sine and cosine functions will go exactly though the sampled points up to rounding errors.

INVERSE DISCRETE FOURIER TRANSFORM

- The equation to get y_n from c_k is called the inverse discrete Fourier transform or inverse DFT and given by

$$y_n = \frac{1}{N} \sum_{k=0}^{N-1} c_k \exp\left(i \frac{2\pi k n}{N}\right)$$

- Starting from a set of y_n values one can determine the c_k values and then get back the y_n values. The y_n values are exact (up to rounding error), but the function you would get does not have to be the same as the function that gave the y_n values. That is the exactness is on the discreteness of the transform, only at the x_n values used will they agree. Multiple different functions that have the same y_n values will have the exact same DFT.

DISCRETE FOURIER TRANSFORM

- If our starting function is real then we gain an additional simplification.
- In that case the C_{N-r} coefficient is equal to the complex conjugate of the c_r coefficient.

$$C_{N-r} = c_r^*$$

- So we only have to calculate coefficients between 0 and N/2. If the function f(x) is complex then we need to calculate all N coefficients.
- The DFT is straightforward to calculate in Python as shown by this example code.

DFT EXAMPLE CODE

```
from numpy import zeros
from cmath import exp,pi
def dft(y):
    N = len(y)
    c = zeros(N//2+1, complex)
    for k in range(N//2+1):
        for n in range(N):
            c[k] += y[n]*exp(-2j*pi*k*n/N)
    return c
```

cmath not math because need complex math

In Python imaginary is j not i

POSITIONS OF THE SAMPLE POINTS

- We can shift the locations of the points where we evaluate f(x) from x_n to $x'_n = x_n + \Delta$. All this will effect is the value of the function at those points y_n. The formula will be essentially the same.
- So far we have been considering points starting at 0 and going to L. This is called at Type-I DFT.
- Alternatively we could take the midpoints of all those points. This is called a Type-II DFT.
- The values of the coefficients will change in the two cases, but otherwise they are equivalent in that the inverse DFT will get you back to the functions values at the points you used.

TWO-DIMENSIONAL FOURIER TRANSFORMS

- Functions of two variables f(x,y) can be Fourier transformed as well. Suppose we have an M×N grid of samples y_{nm}. We first perform an ordinary Fourier transform on each of the M rows.

$$c'_{ml} = \sum_{n=0}^{N-1} y_{mn} \exp\left(-i\frac{2\pi ln}{N}\right)$$

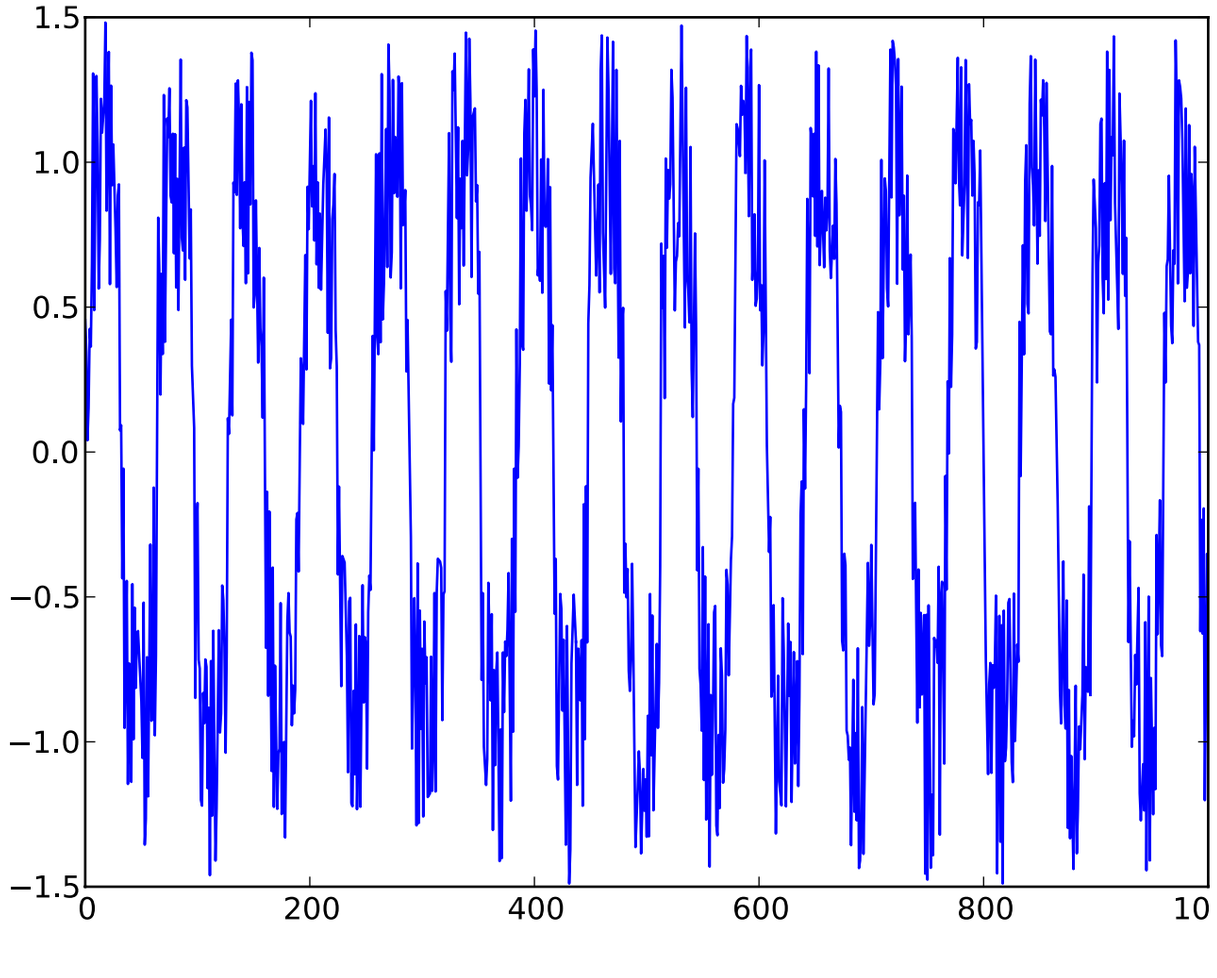
- for each row m we now have N coefficients. Next we take the lth coefficient in each of the M rows and Fourier transform these M values again. As one equation we would have

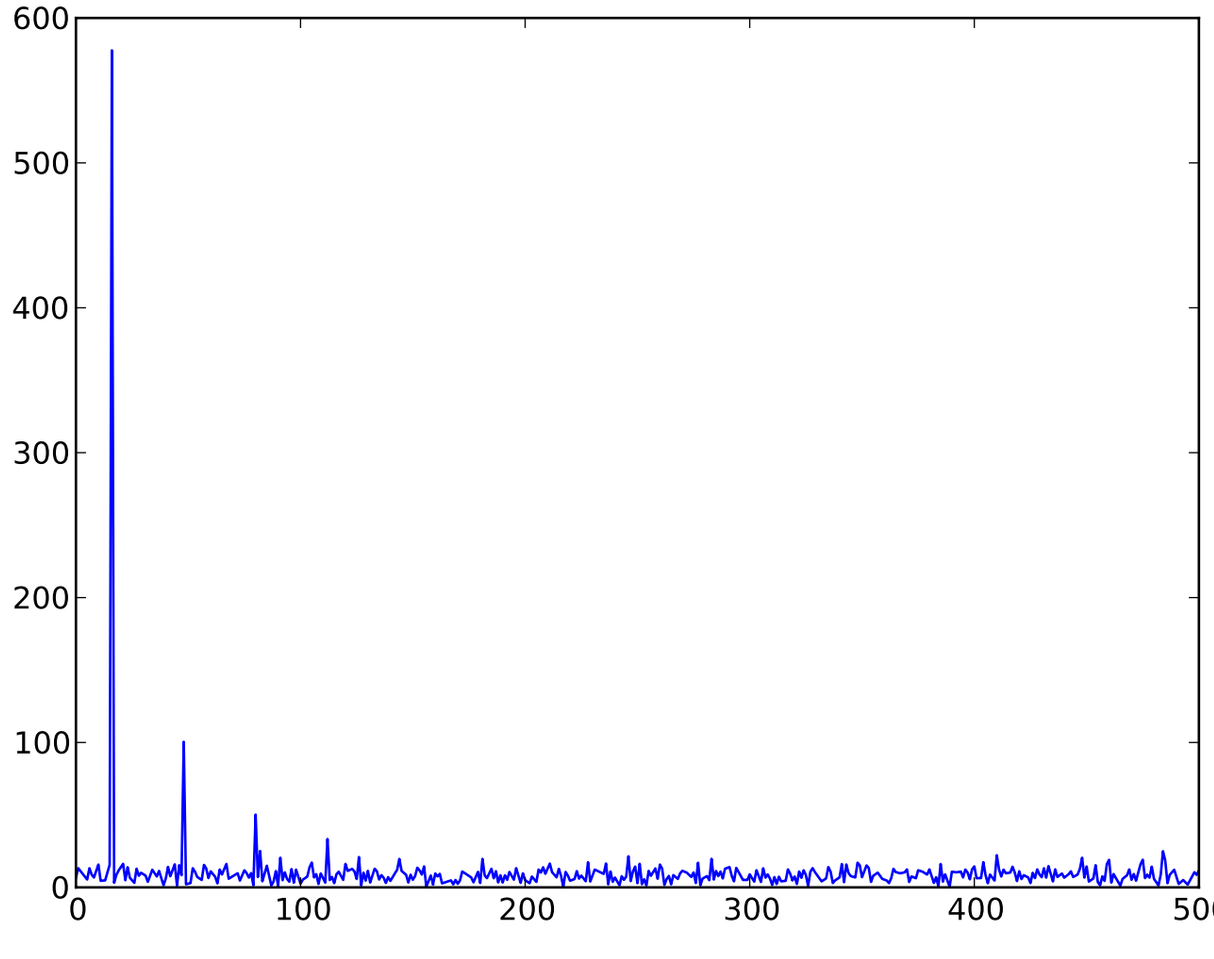
$$c_{kl} = \sum_{m=0}^{M-1}\sum_{n=0}^{N-1} y_{mn} \exp\left[-i2\pi\left(\frac{km}{M} + \frac{ln}{N}\right)\right]$$

- In 2D if our function is real, then the first series we only need up to N/2 because the rest are the conjugates. However these coefficients are complex so we need to evaluate all M.

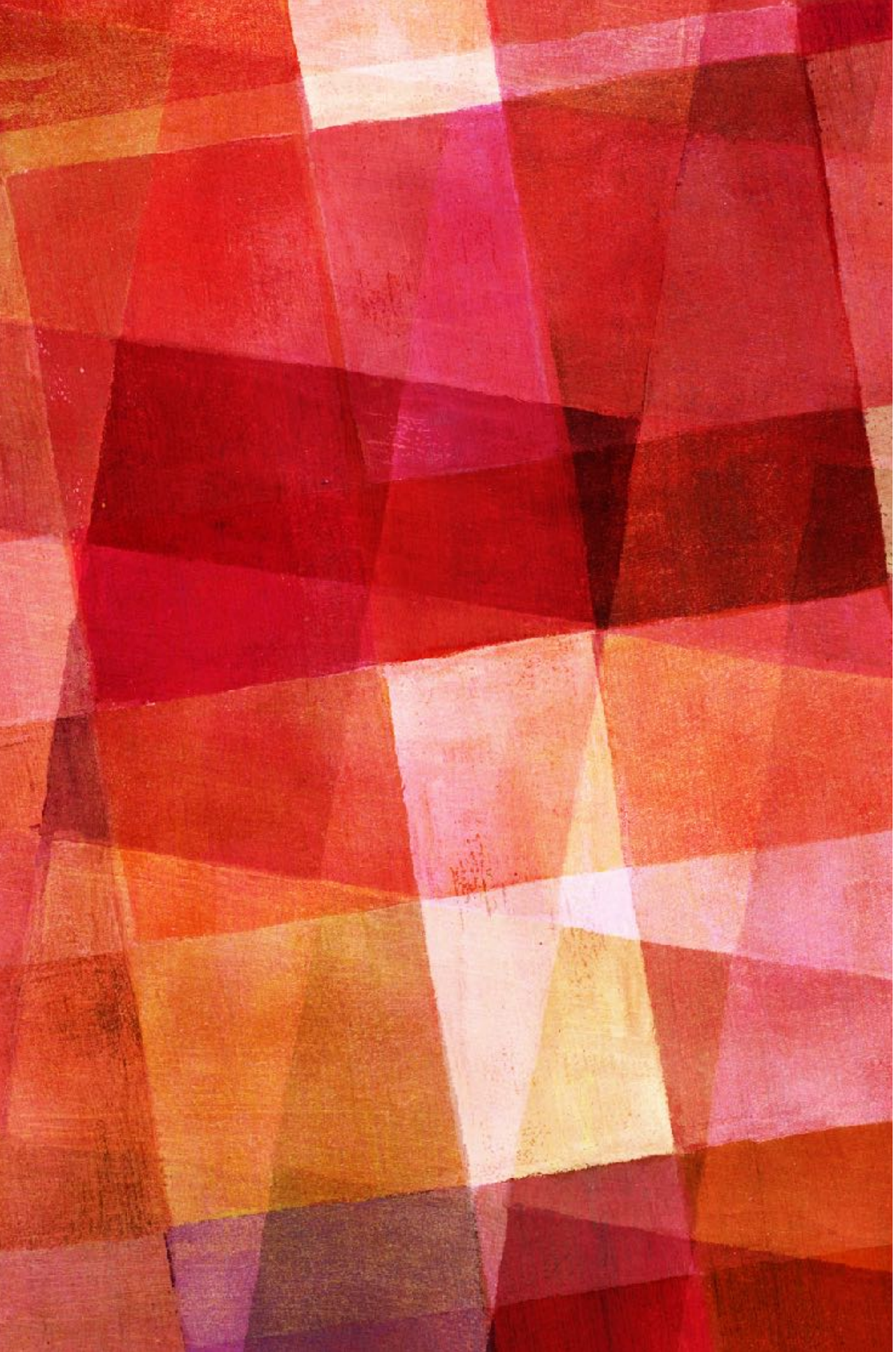
PHYSICAL INTERPRETATION

- It is worth spending some time discussing what a Fourier transform tells us physically about a function.
- The Fourier transform breaks down our function into a series of waves at different frequencies. The coefficients tell us the relative contribution of each frequency. A plot of the absolute values of the coefficients, $|c_k|$, is called a power spectrum and it shows the relative contribution of waves of each frequency.
- We can also remove some of the frequencies, either filtering high frequencies low frequencies or frequencies that contribute little to the overall function. This signal processing can be useful in a number of applications.





A periodic function with some noise. Taking the Fourier transform and plotting it as a power spectrum we can see the signal is primarily at one frequency with some power at harmonics as twice, three and four times the fundamental frequency. We can also see white noise, white because it has roughly the same amplitude at different frequencies. We could filter out the noise by performing the inverse DFT, but only for the 4 or 5 largest Fourier coefficients.



EXERCISE 7.2

- In the on-line resources there is a file called sunspots.txt, which contains the observed number of sunspots on the Sun for each month since January 1749. The file contains two columns of numbers, the first representing the month and the second being the sunspot number.

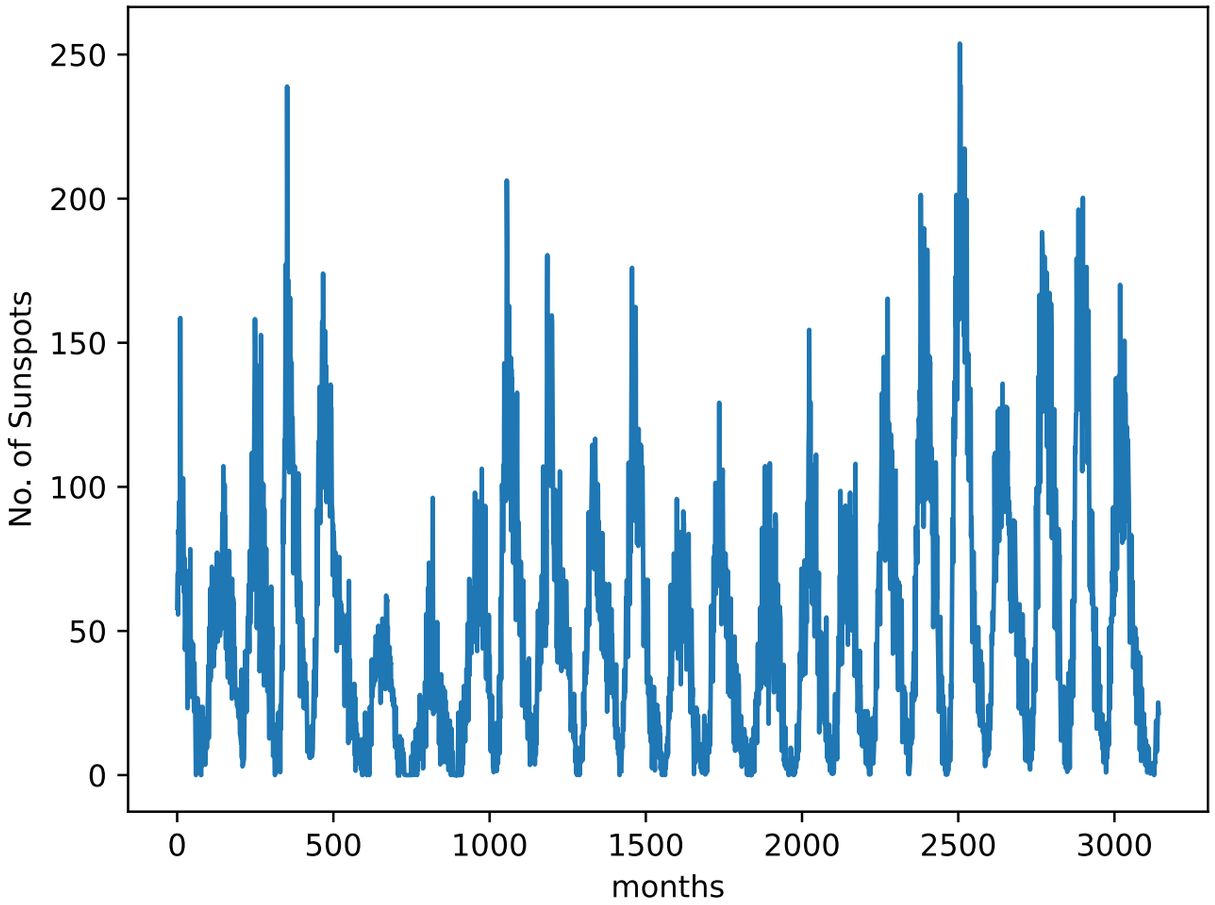
- (a) Write a program that reads the data in the file and makes a graph of sunspots as a function of time. You should see that the number of sunspots has fluctuated on a regular cycle for as long as observations have been recorded. Make an estimate of the length of the cycle in months.

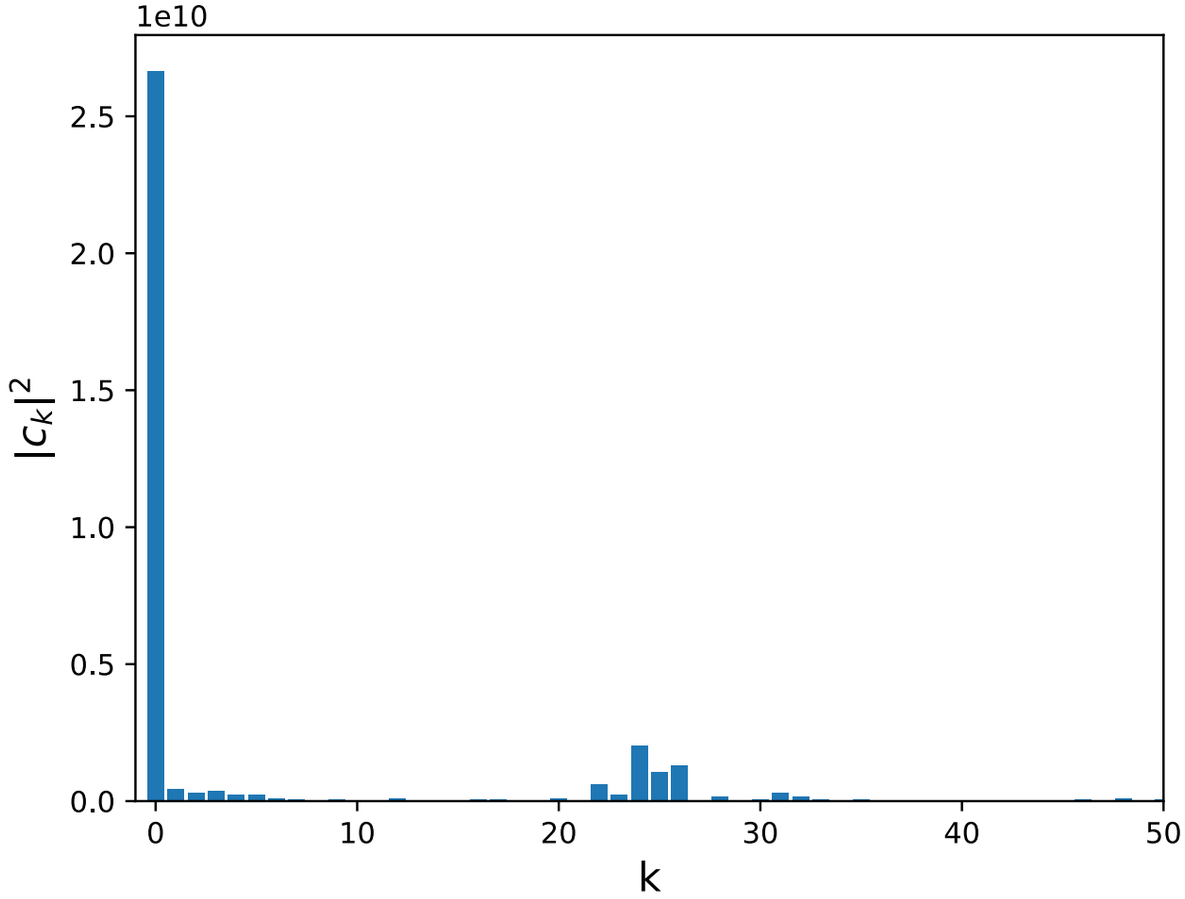
- (b) Modify your program to calculate the Fourier transform of the sunspot data and then make a graph of the magnitude squared c_k^2 of the Fourier coefficients as a function of k also called the power spectrum of the sunspot signal. You should see that there is a noticeable peak in the power spectrum at a nonzero value of k. The appearance of this peak tells us that there is one frequency in the Fourier series that has a higher amplitude than the others around it-meaning that there is a large sine-wave term with this frequency, which corresponds to the periodic wave you can see in the original data.

- Find the approximate value of k to which the peak corresponds. What is the period of the sine wave with this value of k? You should find that the period corresponds roughly to the length of the cycle that you estimated in part(a).

$$c_k = N\gamma_k = \sum_{n=0}^{N-1} f(x_n) \exp\left(-i\frac{2\pi k x_n}{L}\right)$$

MAKING SENSE OF THE POWER SPECTRUM





We can see that the data looks like is varies on some time scale, like ~100 months.

From the power spectrum we see that the k =0 coefficient has 10 times the power as the next coefficient.

THE ZERO TERM

- The inverse transform gives us back our values of f(x) from the Fourier coefficients:

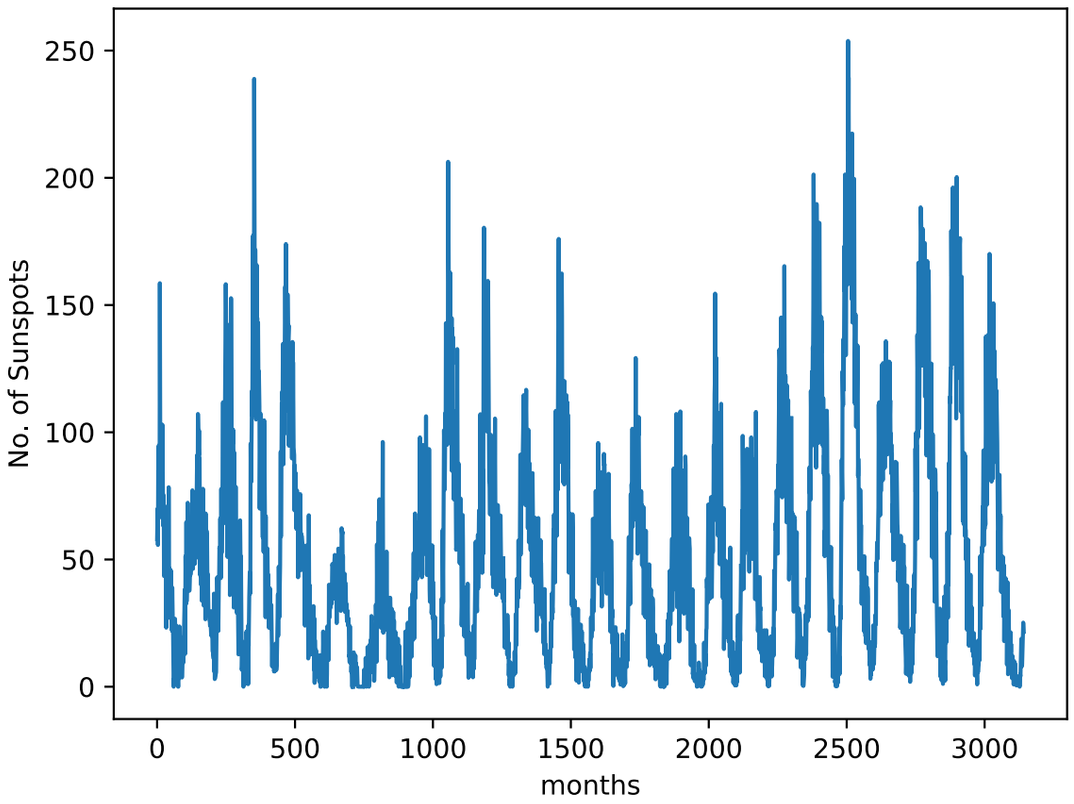
$$y_n = \frac{1}{N} \sum_{k=0}^{N-1} c_k \exp\left(i \frac{2\pi k n}{N}\right)$$

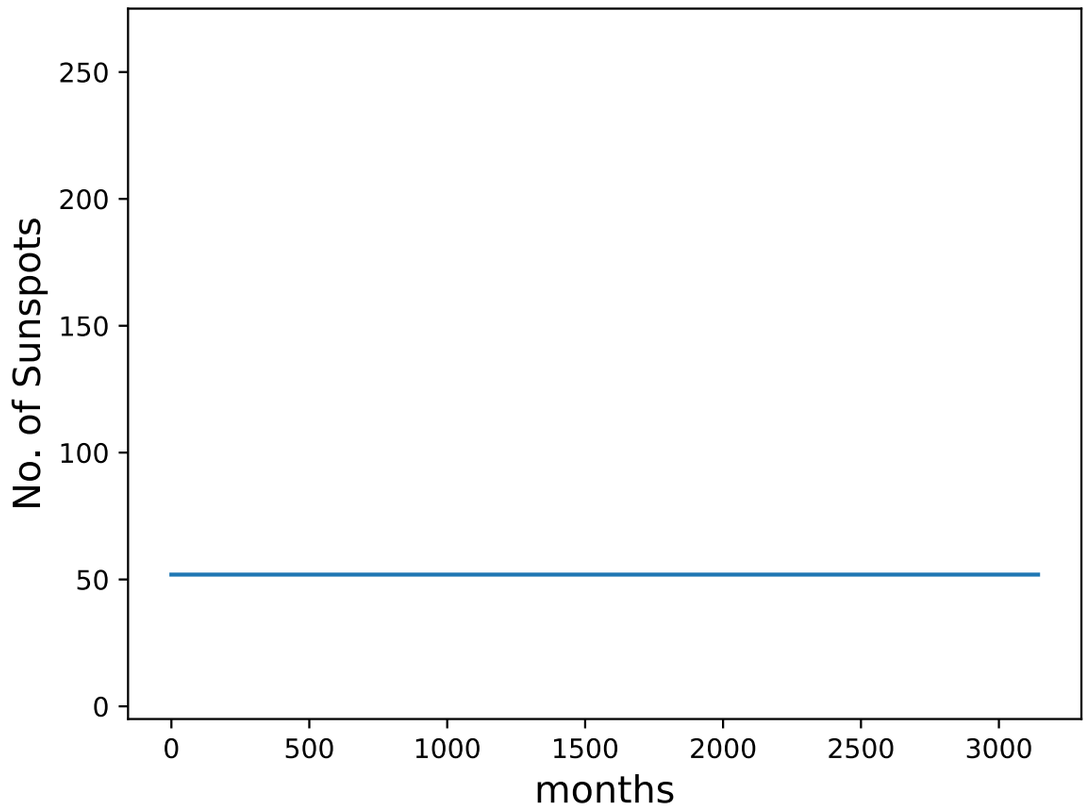
- for the k=0 contribution we get

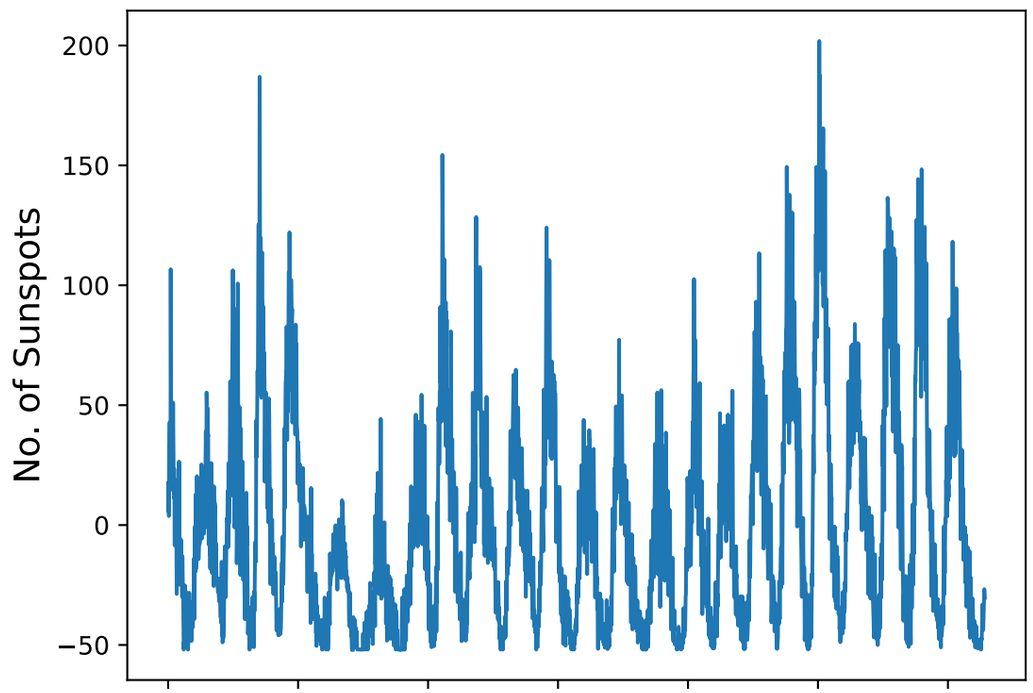
$$y_n = \frac{1}{N} c_0 \exp\left(i \frac{2\pi 0 n}{N}\right) = \frac{c_0}{N}$$

- the same value for all n, in other words the k=0 term is a flat line. This is just the offset of the data from zero, it can be removed by first subtracting this constant value from your data.

THE ZERO TERM

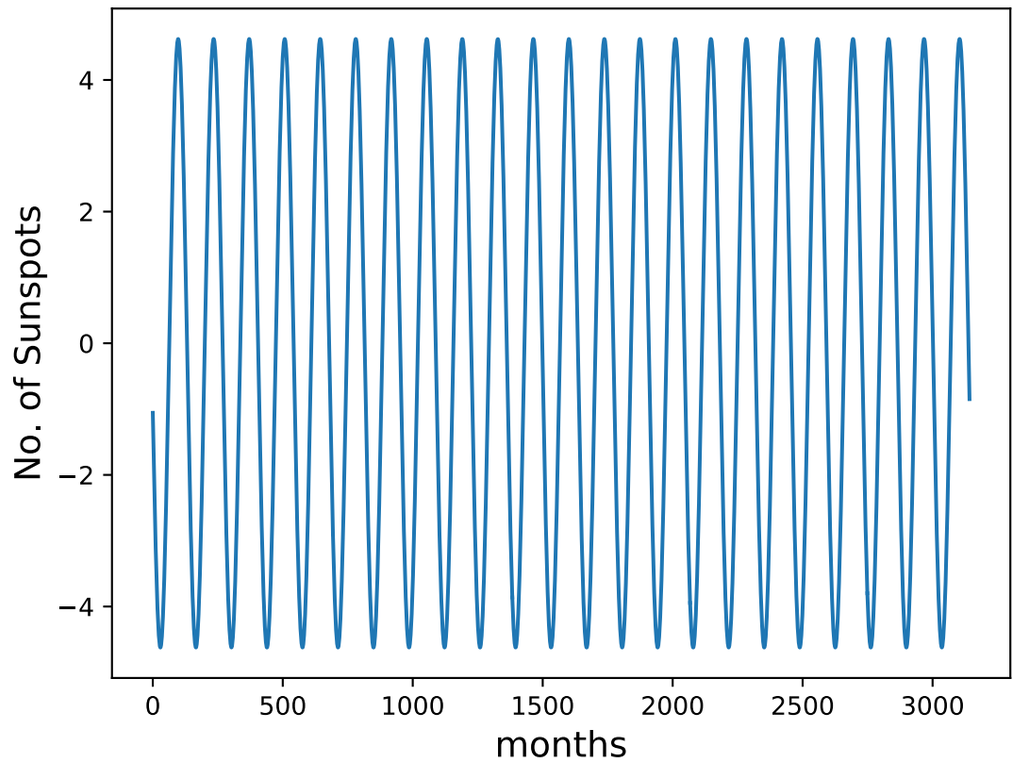


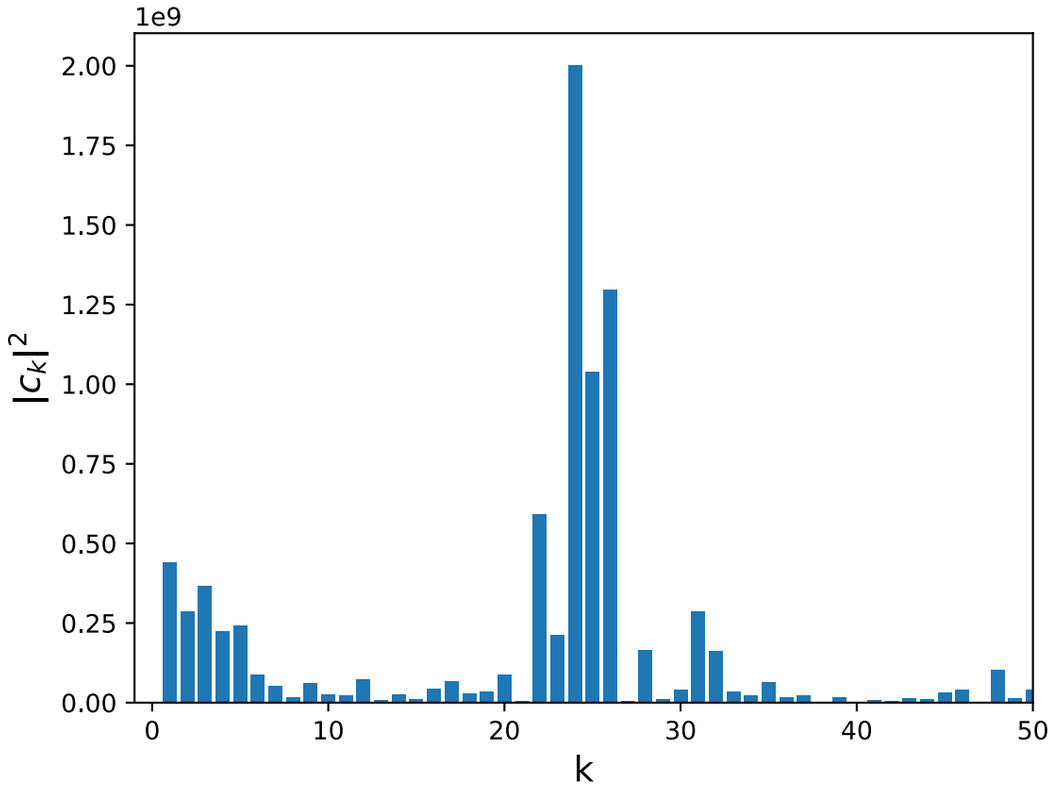


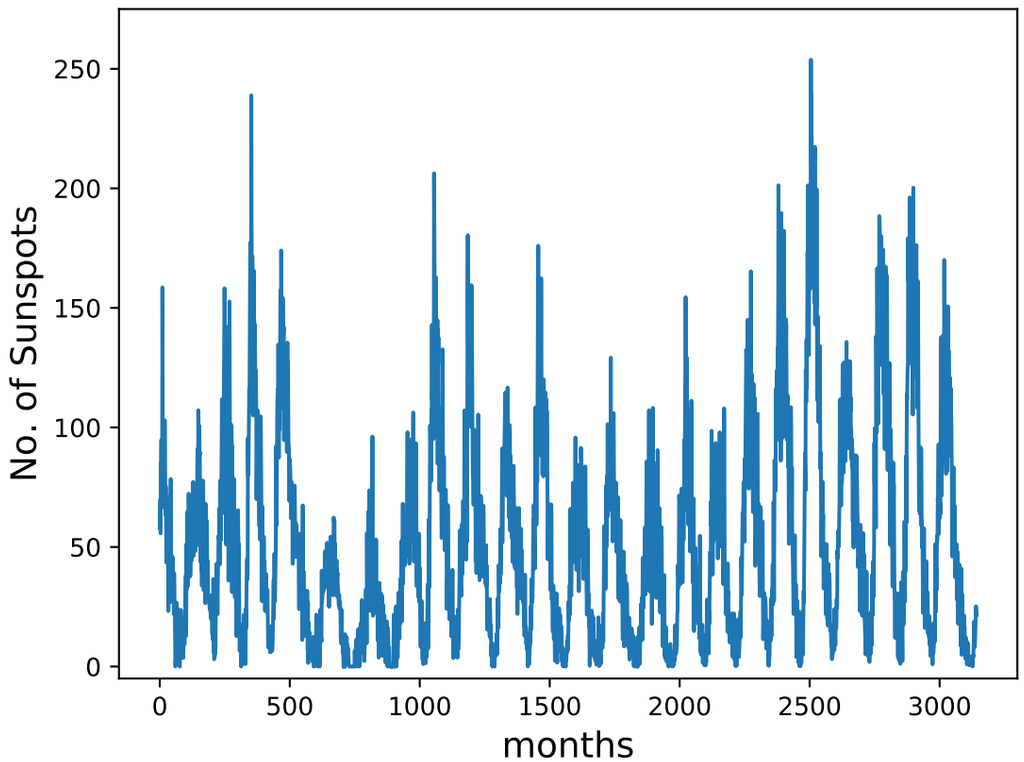


THE K > 0 TERMS

Removing the k=0 term we see the next highest term is k = 23. If we inverse Fourier transform that term we get a simple sine curve. Note that its period is very similar to the period in our data set.

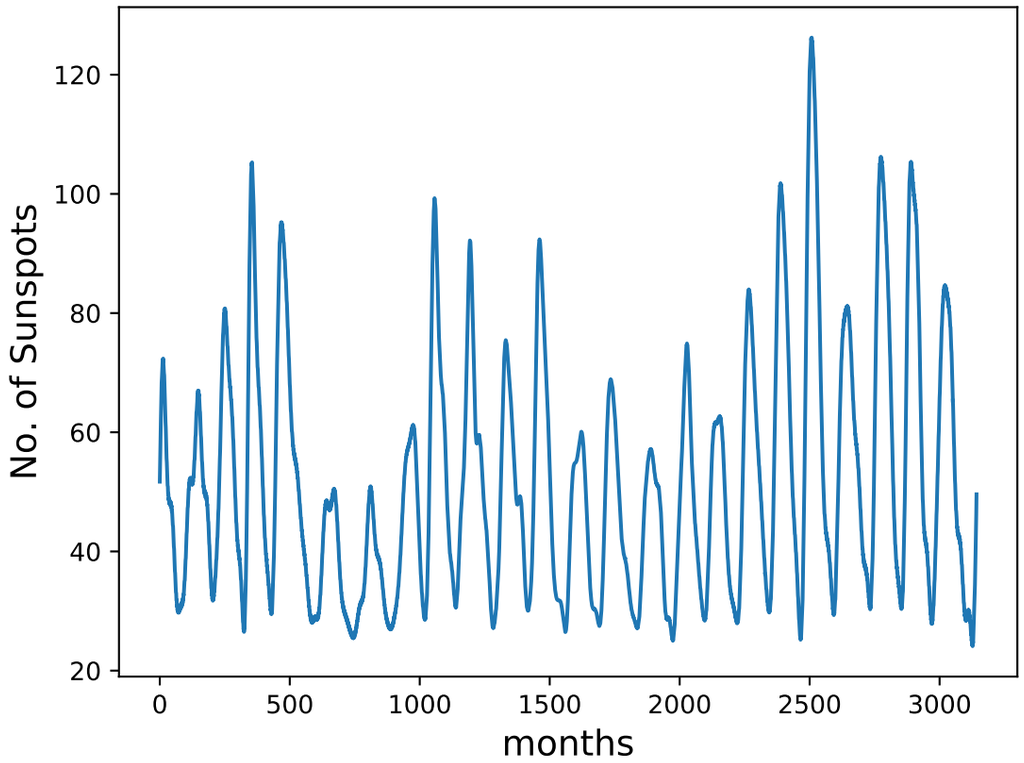


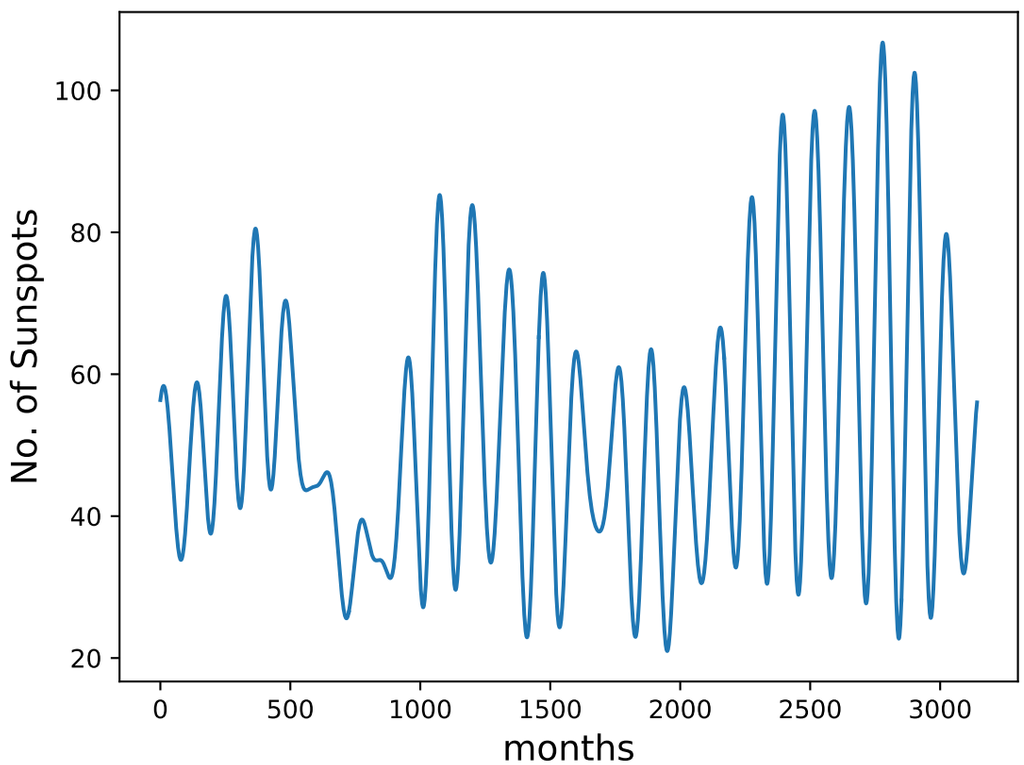


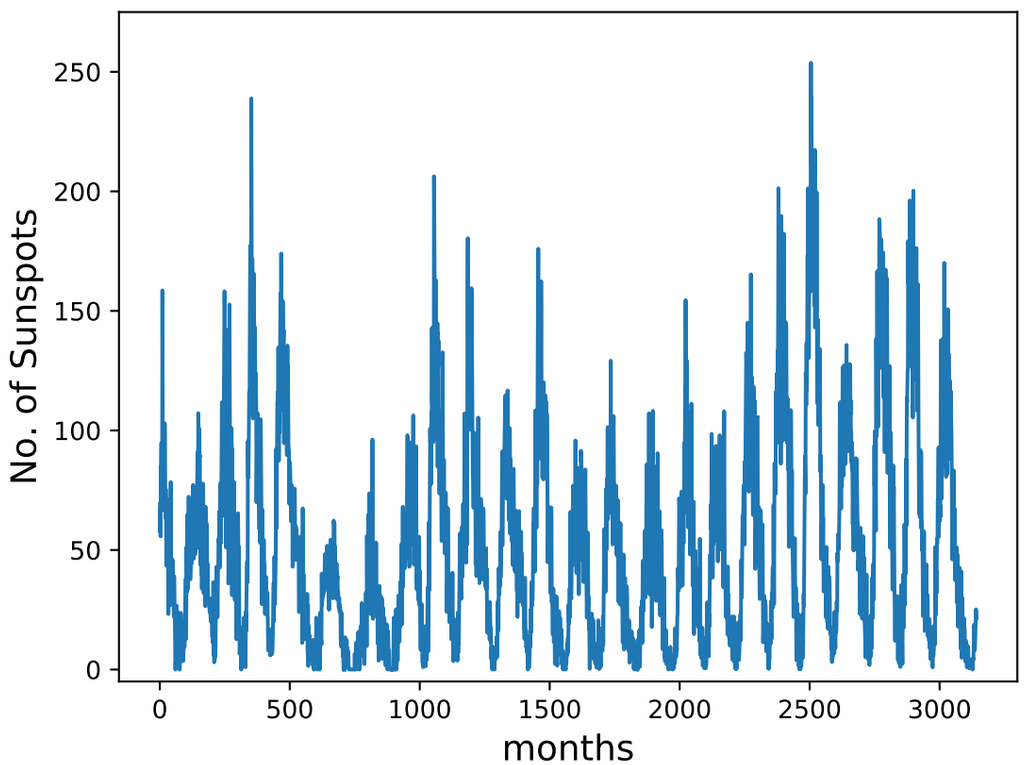


REMOVING THE K=0 TERM

There are about 10 terms with amplitudes that aren't an order of magnitude smaller than $k=23$. If we invert them we get the plot below. If we keep the first 100 terms we get the plot to the right.

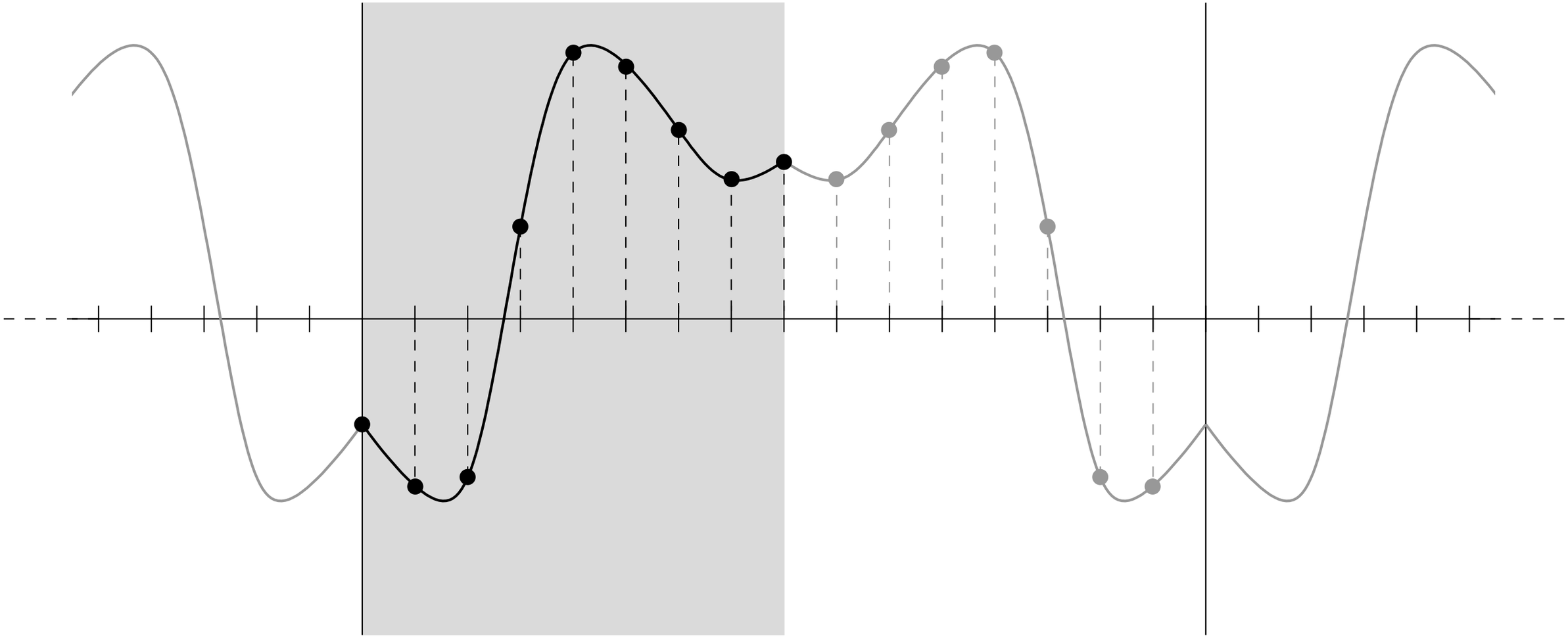






DISCRETE COSINE AND SINE TRANSFORMS

- So far we have been discussing the complex version of the Fourier series, but there is some advantages to thinking about just using the cosines.
- Cosines will only fit symmetric functions, but it is easy to construct a symmetric function by mirroring a function and then repeating.



DISCRETE COSINE AND SINE TRANSFORMS

- While the sine can be used also, because it goes to zero at the end points it is often a poor match to functions of interest.
- The formula will look different depending if we include the endpoints 0 and L (Type-1) or if we take the midpoints (Type-2). For Type-1 we get

$$c_k = y_0 + y_{N/2}\cos\left(\frac{2\pi k(N/2)}{N}\right) + 2\sum_{n=1}^{N/2-1} y_n \cos\left(\frac{2\pi kn}{N}\right)$$

- Notice the sum is only to N/2-1 because the function is symmetric. For the Type-2 points we instead get

$$a_k = 2\sum_{n=0}^{N/2-1} y_n \cos\left(\frac{2\pi k(n+1/2)}{N}\right)$$

- These are referred to as discrete cosine transform or DCT. It is often preferable for data that is not periodic.

TECHNOLOGICAL APPLICATIONS

- While the DFT may seem like something only of special interest to mathematicians and physicist it, it is widely used in common technology.
- For example you are probably aware of the image format JPEG. One way to store the information in a image would be to store a value for each pixel in the image. A jpeg performs a 2D Fourier transform of an image and stores the coefficients. It also doesn't keep some of the smaller coefficients, it this way making the file size much smaller.
- When you load an image your computer performs the inverse DFT to get the image back. There is some information loss because the value of each pixel is not stored, but usually one can't tell any difference by eye.

TECHNOLOGICAL APPLICATIONS

- The MPEG format does the same thing but for movies.
- A similar thing is done for music with the MP3 format, though now the DFT is done in time instead of spatially. The MP3 algorithm is more clever choosing which coefficients to discard based on knowledge of what the human ear can and can not hear.
- For example if there are loud low frequency sounds in a piece of music the ear has a harder time detecting high frequency sounds. So the MP3 format doesn't keep those high frequency coefficients.
- Essentially the entire digital audio and video economy depends on discrete Fourier transforms.

FAST FOURIER TRANSFORMS

- In the DFT we have to perform a sum over N-1 values for 1/2N+1 distinct coefficients. This is $N(1/2N+1) \sim 1/2N^2$ calculations.
- This is not good scaling. If we want to limit ourselves to a billion calculations then we can only have N~45000 sample points. This is only about 1 second of music at today's sampling.
- We are going to need a faster way of performing Fourier transforms if we want to use them on large images, videos and audio files, not to mention computational physics. Luckily Gauss found such a way in 1805 when he was 28 years old.

FAST FOURIER TRANSFORM

- The fast Fourier transform (FFT) is simplest when the number of sample points is a power of two. So let's consider $N = 2^m$.
- We can break the sum of the DFT into a sum over even n and a sum over odd n. The sum of the even terms is then

$$E_k = \sum_{r=0}^{\frac{1}{2}N-1} y_{2r} \exp\left(-i\frac{2\pi k(2r)}{N}\right) = \sum_{r=0}^{\frac{1}{2}N-1} y_{2r} \exp\left(-i\frac{2\pi kr}{\frac{1}{2}N}\right)$$

- but this is just another expression for a Fourier transform, now but for half the number of points. Similarly the odd terms can be written as

$$\sum_{r=0}^{\frac{1}{2}N-1} y_{2r+1} \exp\left(-i\frac{2\pi k(2r+1)}{N}\right) = e^{-i2\pi k/N} \sum_{r=0}^{\frac{1}{2}N-1} y_{2r+1} \exp\left(-i\frac{2\pi kr}{\frac{1}{2}N}\right) = e^{-i2\pi k/N} Q_k$$

FAST FOURIER TRANSFORM

- So we can express the Fourier coefficients as

$$c_k = E_k + e^{-i2\pi k/N} Q_k$$

- The coefficients are just give by the sum of two coefficients determined from a Fourier transforms with half as many points. Plus a term called a *twiddle factor*.

- But E_k and Q_k can just be expressed as the sum of two other Fourier transforms, which can also be broken into two and so on until Fourier transform has only one term. At which point the transform is trivial.

$$c_0 = \sum_{n=0}^{0} y_n e^0 = y_0$$

FAST FOURIER TRANSFORM

- So in practice the fast Fourier transform works by evaluating the coefficient for a single point. Then for two points, four, … until you get the transform for the whole function.
- Thus one ends up needing $N\log_2 N$ calculations instead of $1/2N^2$. This ends up making a big difference. If we have a million sample points the brute force way would require 5×10^{11} calculations while the fast Fourier transform can do it in 2×10^{7}.
- While we have described the algorithm in the sample number is 2^m, it can also be done for any N, but the algebra is more complicated.

STANDARD FUNCTIONS FOR FFT

- Of course since the FFT is so important in computer science there will already exist an implementation in any computer language.
- In Python these live in numpy.fft. The function rfft() will return the coefficients for a set of real sample points while fft() will perform the calculation for a complex set of sample points.
- Note that the array returned by rfft() will only have N/2+1 elements since numpy knows you can get the rest by calculating the complex conjugates.
- To perform the inverse FFT you can use irfft() or ifft().