Fourier analysis

This week, we will learn to apply the technique of Fourier analysis in a simple situation. Fourier analysis is a topic that could cover the better part of a whole graduate level course, if we explored it in detail, but it's also something where in a single lesson, along with a bit of the work from the student, you can go from zero knowledge to being able to do some useful things – you will be exposed to the Fourier transform here, and will develop a tool you can use to compute Fourier transforms, but you should expect that you will have to do a lot more learning if, e.g. you run into a research problem that requires a deep knowledge of Fourier transforms. The Fourier transform is one of the most commonly used mathematical procedures across a wide range of scientific and technical disciplines. It is a process which converts a function into a set of sines and cosines.

Often, we perform what is called a discrete Fourier transform. A function is a continuous set of values. Of course, no experiment will give us a truly continuous set of values, so instead, we will want to have a recipe that can take a set of discrete values, and find the set of functions that describe it. There will be some limitations to the process (e.g. we will not be able to figure out how something is varying when it varies much faster than speed at which we collecting the data).

I will give examples based on the assumption that some quantity is varying in time, but the math can be applied to spatial variations (or variations with respect to some abstract quantity) as well.

The simple, intuitive approach

There is a simple approach to figuring out what a Fourier transform means, which is something that can be used to compute a Fourier transform. There is a mathematical approach called *matched filtering* which can be used in its very simplest form here.

Let's consider some arbitrary function, $f(x)$. It ranges from -1 to $+1$ in value. How can we figure out what it is? We can take some other function whose value we know, and we can multiply the two functions together, and take the integral of the result. If the two functions are the same, then the value of the new function is $f^2(x)$. If the two functions are different the value will be something smaller. This is a trivial case, of course, because if we already know the function $f(x)$, we wouldn't try to represent it as a single function.

Let's consider a slightly different case, that we have a function $f(x)$ which we think is made of the sum of a set of sine waves (or, equivalently, with just a phase shift, cosine waves), but we don't know what the frequencies, normalizations or phases of the sine waves are. Then we wish to multiply the function by every sine wave possible, and see where the integrals come out big and where the come out small. This seems like a daunting job, especially when we consider the infinite number of phases we'd need to check.

We can actually get around the phase problem simply by using a little bit of high school trigonometry.

$$\sin(a+b) = \sin a\cos b + \sin b\cos a. \tag{1}$$

Thus we can take:

$$\sin(2\pi ft+\theta) = \sin(2\pi ft)\cos\theta + \sin\theta\cos(2\pi ft). \tag{2}$$

We can then see that our original function can be expressed as a sine component with a particular frequency and a cosine component with the same frequency, and the relative normalizations of the two components will determine the phase. The phase issue is thus easily converted from an infinite set of possibilities, to just the requirement that we check the normalizations of all the sines and all the cosines.

Thus, for analytic functions we can produce an integral over the sines and an integral over the cosines, and find the coefficients of the sines and cosines. I will spare you the detailed math on this, since it's not the normal way of doing things.

Complex analysis

Instead, combinations of sines and cosines are most conveniently dealt with as complex numbers. A complex number can be written as $re^{i\theta}$, where r is the radius in the complex plane, and θ is the phase in the complex plane. In this way, the real component is $r\cos\theta$ and the imaginary component is $r\sin\theta$. It turns out that this approach leads to a far more efficient way for a computer to deal with taking Fourier transforms than other approaches do.

For taking the Fourier transform, what we want to do is to integrate our function times $e^{i2\pi ft}$:

$$S(f) = \int_{-\infty}^{+\infty} s(t)e^{i2\pi ft}. \tag{3}$$

Then, the real part of the output will give the coefficients of the cosines, and the imaginary part will give the coefficients of the sines.

So this method is fine for getting an idea of what the Fourier transforms of theoretical functions look like, but it's an integral from $-\infty$ to $+\infty$, so we can't use it for working with data. Real data will have a finite duration, and a finite sampling rate. What we want for working with data is to have a methodology that works for discretely sampled, finite length data sets.

To deal with the discrete sampling, what we do is to assume that the function can be represented by a finite number of sines and cosines. For evenly sampled data sets, this leads to the Nyquist frequency – the maximum frequency that we can deal with, which is half of the maximum frequency in the data set, because we need to deal with sines *and* cosines. That is to say, if we have time resolution of one millisecond (i.e. a sampling rate of 1000 Hz), we can measure variations with frequencies no larger than 500 Hz. We also cannot measure frequencies lower than $1/T$, where T is the length of the data set. As a result, we have N data points, and are, in essence, fitting $N/2$ amplitudes and $N/2$ phases to the data. We call this a discrete Fourier transform.

Discrete Fourier transforms

A discrete Fourier transform is a transformation from a list of numbers to a list of amplitudes and phases of sine/cosine waves at different frequencies. The mathematical formulation is as follows:

$$X_k = \sum_{i=0}^{N-1} x_n e^{-i2\pi kn/N}, \tag{4}$$

where we have a series of N values of x_0 to x_{N-1} which are evenly spaced, and wish to get the coefficients for the sines and cosines at frequency k. If we have real numbers only, we can show that we get no information from the last $N/2$ frequencies, so that we do not need to compute them. This expression above, then, is something that we could do on the computer. There is a process for setting up complex variables in C++, but in my opinion, it's not any easier than just putting the real part of the complex number into one variable and the complex part into another array, so I'm not going to spend time teaching it. You are welcome to learn it on your own and use it if you want to give it a try.

Also, it will turn out that the obvious way to turn equation 4 into a computer program turns out to be a really bad way to do it – it gives the right answer, but it is a factor of a few hundred slower than it needs to be.

Practical application on a computer

Now that we have been through some of the theory of Fourier transforms, we can consider how we will do them on the computer. Computer scientists, and scientists who use computers, often like to describe how fast an algorithm is based on how the number of operations on the computer scales with the number of data points, or steps of some other kind that will be used. Constant factors are ignored in this kind of analysis – the difference between taking $2N$ steps or N steps isn't what we concern ourselves with, but rather the difference between, for example, taking N steps and N^2 steps. The former causes us to have to wait a bit longer for the program to run, while the latter can leave us with a very limited range in the size scale of the problem we can attempt to solve.

Suppose we are applying some operation to all the elements on a line segment. That operation would take an amount of computer time proportional to N where there are N elements on the line segment. Suppose further that we were applying the operation to a square made of N line segments in each dimension – then we would end up with a process that takes N^2 time steps. If we then went into even higher dimensions, then we would see that the process goes as N^n, where N is the number of elements per dimension, and n is the number of dimensions. This will be an important motivation for learning to do Monte Carlo integration, a topic we will cover in a few weeks.

The Fourier transform would appear to be something that would take N^2 steps – we have to consider N data points and N frequencies. An clever algorithm, called the Cooley-Tukey algorithm was published in 1965 allowed the computation of Fourier transforms in $N\log N$ steps. We won't go into the details, but we will use the method. The original Cooley-Tukey algorithm works only for data set sizes which are powers of 2. In some cases, this can lead to requiring you to throw away nearly half your data, or to make up fake data points in order to be able to use the rest of your data. There are other algorithms that are a bit more flexible which are based on the same principles, but for this assignment, I will just give you a data set on which to work which has a power of 2 as its number of elements. I will also not explain how the Cooley-Tukey algorithm works, except to say that it essentially divides the data into smaller segments to deal with the Fourier transforms at the higher frequencies.

Using libraries

You should never re-invent the wheel, unless you think you can make a better wheel, or you're doing it as a learning experience. I am thus not going to ask you to write a computer program that implements the Cooley-Tukey algorithm. Instead, I'm going to have you use a routine from the GNU Scientific Library that already does this. We will use the GSL libraries because they allow their libraries to be used freely under all circumstances[1]. There is one catch, which is that you cannot use them in proprietary software.

We will thus have to learn how to use libraries.

This very simple program is taken from the GNU Scientific Library web site:

```
#include <stdio.h>
#include <gsl/gsl_sf_bessel.h>
/*This program has been taken from the GSL web site */
/* http://www.gnu.org/software/gsl/manual/html_node/An-Example-Program.html#An-Example-Progr
*/
int
main (void)
{
double x = 5.0;
double y = gsl_sf_bessel_J0 (x);
printf ("J0(%g) = %.18e\n", x, y);
return 0;
}
```

It includes the library file `gsl_sf_bessel.h` on the second line of the program. There is then a call to a function `gsl_sf_bessel_J0` on a later line, which computes a particular Bessel function's value for $x = 5$. This is a function that we haven't defined – it's within the library file.[2]

There's another step we need to go through for making this work. We need to tell the compile at the time we compile the program that we want to use non-standard libraries. Generally, when you install a library on your computer, you will see some instructions about how to link the libraries at the time you compile. For this one, what you need to do is to add on `-lgsl -lgslcblas -lm`, so you should type:

```
gcc test_gsl.cc -o test_gsl -lgsl -lgslcblas -lm
```

[1]Except in some cases where you want to make money selling the software you have produced using their routines as part of your own code

[2]We've actually been using functions like this all along. If you're curious, or don't believe, me, try running `printf` without putting in `stdio.h`.

at the time you compile. You can also put these commands into a `makefile`, and then just type `make gsl_test` – this is something that you can teach yourself if you're interested. It will take you a bit of time to figure it out, but it will save you a bit of typing. If you get to the point of building complicated programs that involve a lot of subroutines in different files, you eventually need to learn this, but it's not necessary for a first course. If GSL is in a non-standard place on the computer on which you're working, you'll need to do a few other things to make it work.