Project 5: Fourier Analysis

EARS 202

13th February 2025

The method we wish to consider here is not really a statistical method so much as a way of transforming data. Even so, it can be quite useful in statistical and data analysis contexts.

Say we have a dataset our measurements are collected at uniform intervals of some independent variable. In the classic example, this independent variable might be time – so, for example, perhaps we have a set of measurements collected every 0.5 seconds. If we were to just plot our data versus this independent variable, we might see some trends, consider some statistical analyses to perform, etc. However, what if we could consider our data in terms of *frequency* rather of time?

This may be considered a *transform* of our dataset. It may be a bit less intuitive than some other transforms we have previously considered – for example, taking the log of our data (i.e., a *log-transform*) – but at some level both are just different ways of transforming our dataset.

If our data are indeed uniformly spaced in some independent variable domain (time or otherwise), then we can do this with a *discrete Fourier transform*:

$$X_k = \sum_{n=0}^{N-1} x_n \cdot e^{-i2\pi \frac{k}{N} n} \tag{1}$$

for $k \in 0 : N-1$, where the resulting complex numbers X_k represent the amplitude (real) and phase (complex) of a sine wave with frequency k/N. In other words, this transform lets us decompose our signal (any signal!) into a set of sine waves. A number of nice visualizations of this may be found in various places, including some rather fun ones.

In Julia we might consider implementing this (using complex numbers via the `ComplexF64` type) along the lines of:

```
function dft(data)
    N = length(data)
    Xk = Array{ComplexF64}(undef, N)
    for k in 0:N-1
        real = 0.0
        imag = 0.0
        for n in 1:N
            real += data[n] * cos(2pi * k * n/N)
            imag -= data[n] * sin(2pi * k * n/N) * im
        end
        Xk[k+1] = real + imag
    end
    return Xk
```

```
end
```

This should work, but won't scale well to large datasets. Consequently, a more complicated implementation called the *fast fourier transform* is often used in practice. In Julia, this is available using the FFTW.jl package.

1 Applying an FFT in practice

One example dataset we might consider analyzing in this way might be sea level. One such example sea level dataset has been linked on Canvas. You might plot the data and its transform with something along the lines of the following:

```
using FFTW, Plots, StatGeochem

data = importdataset("BarHarborWL2013.csv", ',', importas=:Tuple)

# Signal: water Level
wl = data.Water_Level
# Number of sample points
N = length(data.Hour_Count)
# Sample period
Ts = 24/N
# Start and end time
t0 = 0
tmax = t0 + N * Ts
# Time coordinate
t = t0:Ts:tmax

# Fast Fourier Transform
ft = fft(wl)
F  = fftshift(ft)
freqs = fftshift(fftfreq(N, 1.0/Ts))

# Plot results
time_domain = plot(t[1:end-1], wl, title="Signal", framestyle=:box)
display(time_domain)

freq_domain = plot(freqs, abs.(F), title="Spectrum", yscale=:log10, framestyle=:box)
display(freq_domain)
```

Can you interpret the resulting frequency spectrum? What do the "peaks" in this spectrum represent?

2

Find another timeseries (or other regularly-spaced) dataset of your choice, and conduct a similar analysis as above.

Thinking back to our software engineering lecture, track your code and data with a version control system (e.g. `git`) and turn in your work as a link to a version control server (assuming you don't want to set up your own git server, you can use github, gitlab, sr.ht, etc.).