# Assignment 8 : FFT and the Double Pendulum

due: April 13, 2020 – 11:59 PM

**GOAL:** The first part of the problem is program the general Fast Fourier Transform with complex arithmetic. The second part looks for oscillatory of the double pendulum Fourrier analysis and filtering out high frequency (wiggly) noise by a high frequency cut-off to get a smooth fit.

# I Introduction

The fast Fourier transform is a classic algorithm very important to analyzing signals of all kinds. It was invented by Gauss (who else!) but credited to Cooley and Tukey who rediscovered without knowing Guass' earlier application: https://en.wikipedia.org/wiki/Fast_Fourier_transform

```
Gauss, Carl Friedrich,"Theoria interpolationis methodo nova tractata",
Werke, Band 3,265{327 (Königliche Gesellschaft der Wissenschaften, Göttingen, 1866)

Cooley, James W.; Tukey, John W. (1965). "An algorithm for the machine
calculation of complex Fourier series". Math. Comput. 19:
297{301. doi:10.2307/2003354.
```

This is a classic devide and conquer algorithms doing a specific matrix vector operation in $O(NlogN)$ instead of the straigth forward $O(N^2)$. See the class lecture notes for details.

# II Coding Exercise #1: Program the Complex FFT

The discrete Fourier Tranform on N points and its inverse are

$$f(k) \quad = \sum_{n=0}^{N-1} c_n e^{2\pi ikn/N} c_n \quad = \frac{1}{N} \sum_{k=0}^{N-1} f(k) e^{-2\pi ikn/N} \tag{1}$$

respectively. On GitHUB at `!HW8code/FFT!`there is already code for the slow FT. This exercies is simly to convert it into a FFF.

The deliverables for this exercise are:

- Run the code `!MainFFT.ccp` and verify that the slow ($N^2$) tranforms and its inverse does give the correct result. You can use synthetic daat.
- Add to `MainFFT.ccp` the FFT routines and test them against FT.
- No run speed test of both FT and FFT for large range of sizes $N = 2^n$ with $n = 1, 2, 3 \cdot 10$ at least and plot result to *prove* empirally the scaling of $O(N^2)$ and $O(Nlogn)$ respectively

# III Coding Exercise #2: Discrete Fourier Series Fit.

This problem revolves about the double pendulum [1]. We will analyze the path tracked by a double pendulum and replot it after applying a filter that cuts off high frequencies and compare it with low order polynomial fits to the data. Which is more appropriate? Here we have no error bars—the data is treated as perfect. It is generated by the program `dbl_pendulum.cpp` on GitHub. While you should feel free to play with the program, ultimately your analysis in this assignment should be based on the data in `Trace.dat`. You can plot the data in gnuplot. For example, to plot the sine of the displacement angle, try running the commands:

```
plot [0:1023]    "Trace.dat"  using 1:3 with lines
replot  "Trace.dat" using 1:5  with lines
```

to see the first $2^{10} = 1024$ time slices. There are $2^{13} = 8192$ in the file. The problem is to read this file and fit the first $N = 1024$ data points of the sine of the displacement angle (columns 3 and 5) to a Fourier series. Let $k$ index the $N$ data points, $k = 0, 1, ..., N-1 = 1023$, and $f(k)$ denote the displacement angle for data point $k$. The Fourier series is defined by:

$$f(k) = \sum_{n=0}^{N-1} c_n e^{2\pi ikn/N} = a_0 + \sum_{n=1}^{N-1} \left[a_n \cos(2\pi kn/N) + b_n \sin(2\pi kn/N)\right]. \tag{2}$$

For the right hand side, I have used $c_n = a_n - ib_n$. This formula only works so well for the special case [2] that $f(k)$ is purely real! The $c_n$'s can be defined by:

$$c_n = \frac{1}{N}\sum_{k=0}^{N-1} f(k) e^{-2\pi ikn/N} = \frac{1}{N}\sum_{k=0}^{N-1} f(k)[\cos(2\pi kn/N) - i\sin(2\pi kn/N)] \tag{3}$$

[1] By the way the double pendulum this is a simple example of a chaotic system. (see https://youtu.be/PrPYeu3GRLg.) To see a beautiful video why this relevant to weather see https://youtu.be/aAJkLh76QnM

[2] To see this take the real part of RHS of Eq. 2: $\frac{1}{2}(c_n e^{2\pi ikn/N} + c_n^* e^{2\pi ikn/N}) = \frac{1}{2}(c_n + c_n^*)\cos(2\pi kn/N) + i\frac{1}{2}(c_n - c_n^*)\sin(2\pi kn/N)$ which implies $c_n = a_n - ib_n$. For real series we have a relation $c_n = c_{-n}^*$ so there are actually only $2N + 1$ parameters on both sides of Eq. 2.

where the $f(k)$ are, again, the data for the sine of the displacement angle in `Trace.dat`. How do you extract $a_n$ and $b_n$ from $c_n$? We told you above! You should take advantage of the C++ complex number class which you can include by using `#include <complex>`. The complex class includes functions to extract the real and imaginary part of a complex number, too.

Okay, well, we've given you some equations. What do we want you to do to them?

In this exercise we're going to implement a simple *low pass filter*. In its simplest form, a low pass filter takes a signal and cuts out any high frequency contributions (above some threshold frequency). This is important in audio processing, for example: as a simple example, the human ear can't hear any frequency over 20kHz. If you're compressing audio (say an mp3), what's the point in saving any data corresponding to frequencies above 20kHz? Thus, use a low pass filter and get rid of it!

To implement a low-pass filter, you should follow these steps:

1. Use the function you defined for the first part of the exercise to convert $f(k)$ to frequency, $a_n$ and $b_n$.
2. Zero out an appropriate set of $a_n$ and $b_n$'s corresponding to high frequencies. This may give something away: If you wanted to remove the top half of the frequency space, you'd set $a_n$ and $b_n$ to zero on the range $N/4 < n < 3N/4 - 1$.
3. Transform back using equation 2, plugging in the modified $a_n$ and $b_n$. You should implement the *inverse Fourier transform* in a separate function as well.

You should compare how the data looks as you apply a more and more aggressive low pass filter. Compare, for example:

- The original data $f(k)$, where $f(k)$ is the first 1024 data points in column 3 or 5 of `Trace.dat`.
- The data after removing the top half of the frequency space.
- The data after removing the top 75% of the frequency space.
- The data after removing the top 87.5% of the frequency space.
- ...And so on.

Make some plots and submit a brief qualitative write-up on how the data looks after applying a more and more aggressive low pass filter. A leading question: at what point does the low pass filter start looking bad?

The deliverables for this exercise are:

- At least one source file, `lowpass.cpp`, which prints to file the data in column 3, and to a separate file the data in column 5, before and after applying the low-pass filters described above. In each file, the first column should be the time (which is column 2 of `Trace.dat`), then the subsequent columns should be the values of $f(k)$ for increasingly aggressive low pass filter. Feel free to split the code into multiple files as you see fit—bear in mind that we'll be revisiting Fourier transforms in upcoming assignments, so the more effort you put into writing clean code now, the less pain you'll go through later! Don't forget a makefile, too.
- Plots and a write-up for the first part of the assignment where you describe where the redundancy is in the discrete Fourier transform—the plots should support your write-up! Feel free to make `lowpass.cpp` also print out a file containing the $a_n$'s and $b_n$'s.
- Plots and a write-up for the second part of the assignment where you describe the effect of a low-pass filter as you remove more and more frequencies.

# IV Extra Credit & Extra Fun

First redo the fit in **Coding Exercise #2** above by selecting a few dominate modes in your low pass filter and minimizing the $\chi^2$. Since the data is very good you may set $\sigma's = 1$ through out.

Also you might show the filter on dominate modes in the full FFT is NOT identical to the $\chi^2$ fit of these same modes. Curious?