Curve Fitting: Polynomial vs Fourrier Modes

GOAL: The first part of the exercise includes understanding how to fit noisy data with a low order polynomial to tease out the general trend. The second part looks for oscillatory behavior using Fourrier analysis and again filtering out high frequency (wiggly) noise by a high frequency cut-off.

seehttps://en.wikipedia.org/wiki/Overfitting

https://en.wikipedia.org/wiki/Lagrange_polynomial

I Example #1: Polynomial Fits and Error Estimate

Recall in class we used gnuplot to show that temperature is rising in the data for the interval [1900 : 2015] with the commands:

```
plot "Complete_TAVG_summary.txt" using 1:2:3 with errorbars
f(x) = a + b*(x-1750)
fit [1900:2015] f(x) "Complete_TAVG_summary.txt" using 1:2:3 via a,b
replot f(x)
```

This exercise is to write your own program to do a polynomial fit to a discrete data file with N_0 values y_i, x_i, σ_i for $i = 0, 1, 2, \cdots N_0 - 1$. With errors (e.g. σ_i) you should calculate the χ^2 of our fit to see if you are able to reliably extract information from the fit.

You should use the file `Complete_TAVG_summary.txt` to test of the code. The problem here is to fit the N_0 data values in the time interval [1900 : 2015] of the file `Complete_TAVG_summary.txt`. The program should allow for any reasonable polynomial fit to $y = f(x) = c_0 + c_1 x + c_2 x^2 + \cdots c_N x^N$ for $N = 0, 1, 2, .., N_0 - 1$

1. First fit the data with an exact fit ($N + 1 = N_0$, $\chi^2 = 0$) given by:

$$f(x) = \sum_{j=0}^{N_0-1} y_j \prod_{i \neq j} \frac{x - x_i}{x_j - x_i} \tag{1}$$

   (You should avoid doing this product with $O(N^2)$ operations! There are references online on how to do this.)

2. Next take the number of parameters to be much less than N_0 ($N \ll N_0$) and get a least square fit. Here don't let N be too large. Only try $N = 0$ (constant), $N = 1$ (linear), $N = 2$ (quadratic), and $N = 3$ (cubic). The chi square is defined to be

$$\chi^2 = \sum_{i=0}^{N_0} \frac{(y_i - f(x_i))^2}{\sigma_i^2} \tag{2}$$

The reduced chi square is

$$\chi^2_{reduced} = \chi^2/(N_0 - N - 1) \tag{3}$$

It tells you how well on average the curve goes through the error bars for each point x_i, remembering that given $N+1$ parameters chi-square would be zero for that many points.

We can find the values of $c's$ that minimize the chi-square by solving the N+1 linear equations [1],

$$\sum_{m=0}^{N} A_{nm} c_m = b_n \tag{4}$$

defined by

$$A_{nm} = \sum_{i=0}^{N_0-1} \frac{x_i^{n+m}}{\sigma_i^2} \quad , \quad b_n = \sum_{i=0}^{N_0-1} \frac{x_i^n y_i}{\sigma_i^2} \tag{5}$$

for all $n, m = 0, 1, 2, ..., N$. You can solve this linear system with Gaussian elimination using the code from Problem Set #4.

The deliverables for this exercise are:

- A fit to the entire data set in the range [1900 : 2015] using Eq. 1.This may be expensive; you can use a smaller range: 32 to 64 points is ok. Submit a plot of the fit against the data.
- A fit from minimizing a χ^2 for $N = 0, 1, 2, 3$. Plot each fit against each other and report the value of χ^2 and $\chi^2_{reduced}$.
- You will probably want to use a Makefile as described in part 3 of this assignment.

II Example #2: Discrete Fourier Series Fit.

This problem revolves about the double pendulum [2]. 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

[1]By the way the proof of Eq.4 take one line of algebra. Namely it is equivalent to setting all derivatives with respect c_n,

$$\frac{1}{2}\frac{\partial \chi^2}{\partial c_n} = -\sum_{i=0}^{N_0} \frac{x_i^n(y_i - \sum_m c_m x_i^m)}{\sigma_i^2} = \sum_m \sum_{i=0}^{N_0} \frac{x_i^n x_i^m c_m}{\sigma_i^2} - \sum_{i=0}^{N_0} \frac{x_i^n y_i}{\sigma_i^2} = 0 \ .$$

to zero! See Lecture on `Curve Fitting` for more details. If you start with an over complete constraints trying to fit, $Mc \simeq y$, where there is a small vector of constants $c = \{c_0, \cdots, c_N\}$ and a large number of measurements $y = \{y0, \cdots, y_{N_0-1}\}$. M is $N \times N_0$ non-square matrix. The the least square problem is give by the linear algebra problem $Ac = b$ for the $N \times N$ square matrix $A = M^T M$ and $b = A^T y$.

[2]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

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:

$$f(k) = \sum_{n=0}^{N-1} c_n e^{2\pi ikn/N} = a_0 + \sum_{n=1}^{N-1} \left[a_n \cos(2\pi kn/N) + b_n \sin(2\pi kn/N)\right]. \tag{6}$$

For the right hand side, I have used $c_n = a_n - ib_n$. This formula only works so well for the special case [3] that $f(k)$ is purely real! The c_n's can be defined by:

$$c_n = \frac{1}{N}\sum_{k=0}^{N-1} f(k)e^{-2\pi ikn/N} = \frac{1}{N}\sum_{k=0}^{N-1} f(k)[\cos(2\pi kn/N) - i\sin(2\pi kn/N)] \tag{7}$$

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 6, plugging in the modified a_n and b_n. You should implement the *inverse Fourier transform* in a separate function as well.

[3]To see this take the real part of RHS of Eq. 6: $\frac{1}{2}(c_n e^{2\pi ikn/N} + c_n^* e^{2\pi ikn/N}) = \frac{1}{2}(c_n + c_n^*)\cos(2\pi kn/N) + i\frac{1}{2}(c_n - c_n^*)\sin(2\pi kn/N)$ which implies $c_n = a_n - ib_n$. For real series we have a relation $c_n = c_{-n}^*$ so there are actually only N real parameters on both sides of Eq. 6. instead of N complex (aka $2N$ real).

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!
- 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.