COE 311K Homework Assignment 3

Due: March 8 2024

Objectives: This assignment should cover the topics of least squares, Fourier series, and interpolation.

Instructions: Complete the following problems 1-6. Each problem is equally weighted 15 points each, and 10 points are given for write-up and code quality for a maximum score of 100. To receive full credit for write-up, scan your problems in sequential order and do not have multiple problems on a single page. To receive full credit for the coding, include comments and be neat with your code. The bonus, if completed all correctly, can add 10 points to your grade for a maximum possible grade of 110. Your homework should consist of 2 files that should be uploaded to Gradescope, a source code file and a "write up" file.

For the coding portions, which are P2,3,4,6. The requested function should be added in a file named HW3.py and submit this as your source code to Gradescope. All functions from those problems should be contained in the same file.

For the rest of the problems (P1,P5,Bonus1,Bonus2) please provide steps, answers, images, screen output, and source code snippets as requested by the problem. For the math problems, feel free to use paper and pencil and scan it. Compile all of this work into a single pdf document called HW3.pdf. Be sure to label which problem is which and separate problems on separate pages.

1 Least Squares

Problem 1

Consider the following set of data:

$$\begin{bmatrix} x & 0.1 & 0.2 & 0.4 & 0.6 & 0.9 & 1.3 & 1.5 & 1.7 & 1.8 \\ y & 0.75 & 1.25 & 1.45 & 1.25 & 0.85 & 0.55 & 0.35 & 0.28 & 0.18 \end{bmatrix}$$

Perform a least squares regression to fit the data to the following curve:

$$y = \alpha_1 x e^{\alpha_2 x}$$

. Be sure to show the following steps on paper:

- linearization of $y = \alpha_1 x e^{\alpha_2 x}$ with respect to the parameters (α_1, α_2)
- Define the sum of squared errors S_r

- Differentiation of S_r with respect to each of the free parameters
- Definition of the normal equations
- your estimate for α_1, α_2

Once these steps have been shown, feel free to code the normal equations in matrix vector form and solve with Python. Be sure to show snapshots of any code as well as the estimates for the parameters α_1, α_2. Finally, generate a plot of the curve fit compared to the given data.

Problem 2

Write a function called `least_squares_poly()` that performs a least squares regression on a general polynomial of the following form and add this to your HW3.py file.

$$f(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3 + ...$$

The function should take the following inputs in this exact order:

- numpy vector x: the vector of data that are the inputs to the polynomial you want to fit
- numpy vector fx: the vector of data that are the outputs you want to fit the polynomial to
- int k: the order of the polynomial that will be fitted, $k \geq 0$

and output the following outputs:

- numpy vector a: the vector of length $k+1$ which contains the coefficients $a_0, a_1, ..., a_k$ which define the polynomial that minimizes the sum of squared error

Hint: The function should take the data and then form the normal equations $Z^T Z a = Z^T b$ to solve for a. You can test your function by generating some data that is close to a certain curve (Ex: $y = 5 + x + 2x^2$) and then see if your code recovers something close to $a = (5, 1, 2)$ in this case.

Problem 3

Write a function called `least_squares_fourier()` that performs a least squares regression on a general Fourier series of the following form and add this to your HW3.py file.

$$\begin{aligned} f(x) = a_0 + a_1 \cos(\omega_0 x) + b_1 \sin(\omega_0 x) + a_2 \cos(2\omega_0 x) + \\ b_2 \sin(2\omega_0 x) + a_3 \cos(3\omega_0 x) + b_3 \sin(3\omega_0 x) + ... \\ + a_k \cos(k\omega_0 x) + b_k \sin(k\omega_0 x) \end{aligned}$$

The function should take the following inputs in this exact order:

- numpy vector x: the vector of data that are the inputs to the Fourier series you want to fit
- numpy vector fx: the vector of data that are the outputs you want to fit the Fourier series to
- int k: the number of frequencies you want in the Fourier series, $k \geq 0$ and note that for a choice of k (besides $k = 0$) you will have both a sin and cos term. So for example if the input is $k = 3$, the unknown coefficients will be $a_0, a_1, b_1, a_2, b_2, a_3, b_3$
- int ω_o: the fundamental frequency of the Fourier series that was shown in the definition of the Fourier series

and output the following outputs:

- numpy vector a: the vector of length $2k+1$ which contains the coefficients $a_0, a_1, b_1, a_2, b_2 ..., a_k, b_k$ which define the Fourier series that minimizes the sum of squared error (output them in the order they appear in the Fourier series).

Hint: You may assume the input data is equally spaced. The function can either form the matrix and vector Z, b as discussed in class and then form the normal equations $Z^T Za = Z^T b$ to solve for a. Or, since you can assume all points are equispaced, use the simplified normal equations as from the lecture notes, either way is OK. You can test your function by generating some data that is close to a certain sinusoidal curve (Ex: $y = -7+2cos(x)-3sin(x)+1cos(2x)+0.5sin(2x)$) and then see if your code recovers something close to $a = (7, 2, -3, 1, 0.5)$ in this case (note in this case $\omega_o = 1, k = 2$).

2 Fourier Analysis

Problem 4

Write a function called `my_dft()` and add this to your HW3.py file. Make the function `my_dft()` so that it performs the Discrete Fourier Transform on a data set of length n, (t_j, f_j) for $j = 0, 1, ..., n-1$. Use the definition of the DFT of the form:

$$F_k = \sum_{j=0}^{n-1} f_j e^{-ik\omega_o j} \quad \text{for k=0,1,..,n-1.}$$

Where the fundamental frequency $\omega_o = 2\pi/n$. The function that you add to your HW3.py script will take the following inputs in this exact order:

- numpy vector f: the vector of data that are the outputs you want to perform Fourier analysis on and outputs the following:

- numpy vector (complex) F: the vector of length n which contains the Fourier coefficients

Hints: Note that the function will take in the data, and compute the coefficients F_k by performing a matrix-vector multiplications. Also observe that the matrix and the output vector will be complex numbers. No system of equations need to be solved, just a matrix vector multiplication. Python has built-in support for complex numbers.

In python, at the k^{th} row and l^{th} column of the matrix you will need (which is complex) may be calculated as: `np.exp(-1j *k * (2* np.pi/ n) * l )` $1j$ invokes the imaginary component for a complex number in Python.

If done properly, the DFT will return a complex vector F where the amplitudes at the k^{th} discrete frequency is given by:

$$amp_k = \frac{\sqrt{Re(F_k)^2 + Im(F_k)^2}}{n}$$

A good example to test your code on in Python is shown in Figure 1. The amplitude info in the output of the DFT is actually redundant and only the first half (or second half) of the output vector contains unique values, this is why in the codes in the Figures I take just the first half of each output vector before computing amplitudes.

Problem 5

Apply your `my_dft()` function from Problem 4 or the built-in `fft()` function in Python to perform a Discrete Fourier Transform on data generated from the function following function:

$$f(t) = 1.5 + 1.8cos(2\pi(12)t) + 0.8sin(2\pi(20)t) - 1.25cos(2\pi(28)t)$$

. To generate the data, take $n = 64$ samples with a sampling frequency of $f_s = 128s^{-1}$. In your code, in addition to computing the DFT, print out the following quantities: $\Delta t, t_n, \Delta f, f_{min}, f_{max}$. Δt is the time period between data points, t_n is the time period covered by the data, Δf is the spacing between frequencies of the DFT, f_{min} is the smallest frequency measured by the DFT, and f_{max} is the max frequency measured by the DFT. Also generate one plot showing the signal $f(t)$ versus time t. Generate one plot showing the amplitude vs frequency taken from the output of the DFT (this was defined in the problem statement for Problem 4).

3 Polynomial Interpolation

Problem 6

Create a function called `my_poly_interp()` which can perform either interpolation with Lagrange polynomial or Newton polynomial as specified by the user.

The function that you add to your HW3.py script will take the following inputs in this exact order:

- numpy vector x: the vector of data that are the inputs for which you wish to interpolate within
- numpy vector *fx*: the vector of data that are the outputs you want to perform interpolation on
- numpy vector *xi*: A vector of points the user wishes to interpolate fx at
- string *ptype*: A string the user can specified where if the user puts "Lagrange" it will do Lagrange interpolation and if the user puts "Newton" it will do Newton interpolation.

and outputs the following:

- numpy vector *fxi*: the vector of the same length as *xi* with the interpolated values at those respective points

Hints: Recall that given data of length n, this will uniquely define a polynomial of order $n-1$ which passes exactly through all n of the points. A good test for your data can be seen in Figure 2.

Bonus

Part 1

Apply your `my_poly_interp()` function to interpolate the Heaviside function defined as:

$$f(x) = \begin{cases} 0 & \text{if} \quad x < 0 \\ 1 & \text{if} \quad x \geq 0 \end{cases}$$

Interpolate using both the Newton and Lagrange polynomials on the range $x = [-1, 1]$ with the following number of equally spaced points: $n = 3, 5, 7, 9$. Plot your results along with a line with the reference solution. Add the snapshot of the code along with the plot to the write up. What do you see to the interpolation as you increase the polynomial order for interpolation?

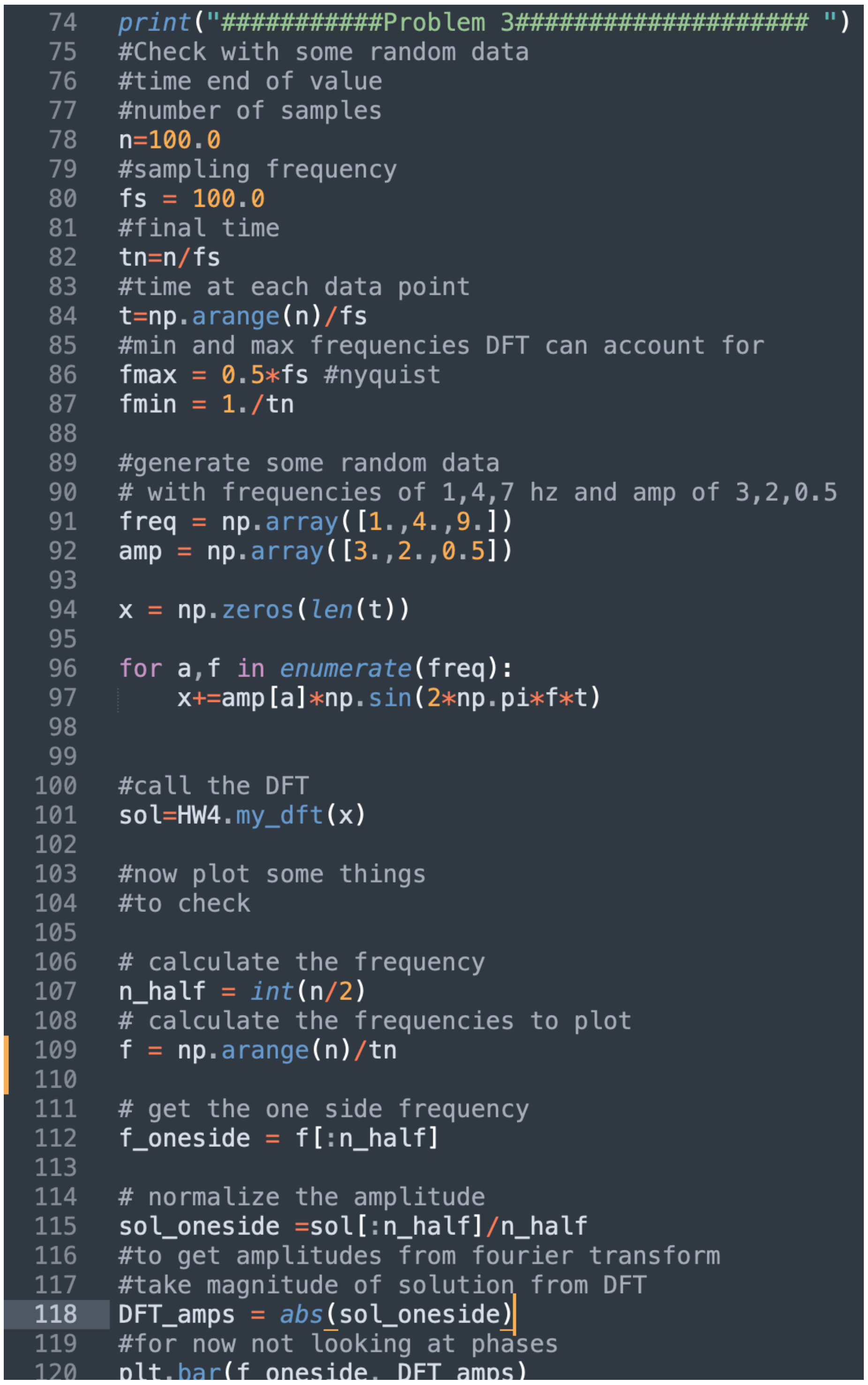
Part 2

Solve the same problem as in Problem 1 but instead skip the manual differentiation of S_r and instead solve the normal equations as defined:

$$Z^T Z\mathbf{a} = Z^T\mathbf{b}.$$

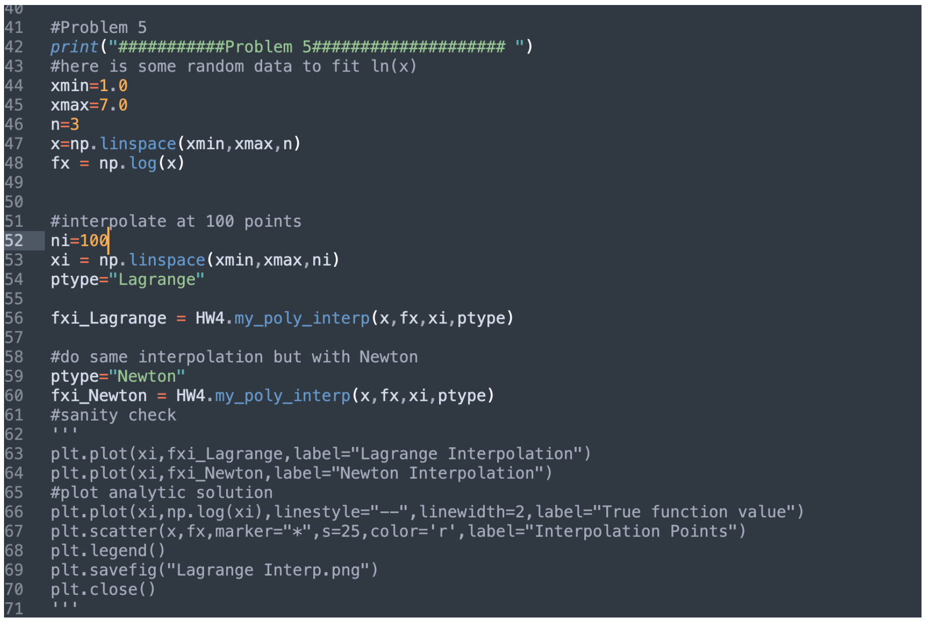
On paper, define what Z is as well as $\mathbf{a}$ and $\mathbf{b}$. Proceed to solve the equations with Python and include snapshots of the code as well as the resulting output $\mathbf{a}$ in your write-up.

4 Figures



```python
print("##########Problem 3################### ")
#Check with some random data
#time end of value
#number of samples
n=100.0
#sampling frequency
fs = 100.0
#final time
tn=n/fs
#time at each data point
t=np.arange(n)/fs
#min and max frequencies DFT can account for
fmax = 0.5*fs #nyquist
fmin = 1./tn

#generate some random data
# with frequencies of 1,4,7 hz and amp of 3,2,0.5
freq = np.array([1.,4.,9.])
amp = np.array([3.,2.,0.5])

x = np.zeros(len(t))

for a,f in enumerate(freq):
    x+=amp[a]*np.sin(2*np.pi*f*t)


#call the DFT
sol=HW4.my_dft(x)

#now plot some things
#to check

# calculate the frequency
n_half = int(n/2)
# calculate the frequencies to plot
f = np.arange(n)/tn

# get the one side frequency
f_oneside = f[:n_half]

# normalize the amplitude
sol_oneside =sol[:n_half]/n_half
#to get amplitudes from fourier transform
#take magnitude of solution from DFT
DFT_amps = abs(sol_oneside)
#for now not looking at phases
plt.bar(f_oneside, DFT_amps)
```

Figure 1: Problem 4 Python Example



```python
#Problem 5
print("##########Problem 5################## ")
#here is some random data to fit ln(x)
xmin=1.0
xmax=7.0
n=3
x=np.linspace(xmin,xmax,n)
fx = np.log(x)


#interpolate at 100 points
ni=100
xi = np.linspace(xmin,xmax,ni)
ptype="Lagrange"

fxi_Lagrange = HW4.my_poly_interp(x,fx,xi,ptype)

#do same interpolation but with Newton
ptype="Newton"
fxi_Newton = HW4.my_poly_interp(x,fx,xi,ptype)
#sanity check
'''
plt.plot(xi,fxi_Lagrange,label="Lagrange Interpolation")
plt.plot(xi,fxi_Newton,label="Newton Interpolation")
#plot analytic solution
plt.plot(xi,np.log(xi),linestyle="--",linewidth=2,label="True function value")
plt.scatter(x,fx,marker="*",s=25,color='r',label="Interpolation Points")
plt.legend()
plt.savefig("Lagrange Interp.png")
plt.close()
'''
```

Figure 2: Problem 5 Python example