C200 Programming Assignment № 8

Dr. M.M. Dalkilic
Computer Science
School of Informatics, Computing, and Engineering

Indiana University, Bloomington, IN, USA

November 12, 2023

The HW is due on **Saturday, November 18 at 11:05 PM EST**. Please commit, push and submit your work to the Autograder before the deadline.

1. Make sure that you are **following the instructions** in the PDF, especially the format of output returned by the functions. For example, if a function is expected to return a numerical value, then make sure that a numerical value is returned (not a list or a dictionary). Similarly, if a list is expected to be returned then return a list (not a tuple, set or dictionary).

2. Test **debug** the code well (syntax, logical and implementation errors), before submitting to the Autograder. These errors can be easily fixed by running the code in VSC and watching for unexpected behavior such as, program failing with syntax error or not returning correct output.

3. Make sure that the **code does not have infinite loop (that never exits) or an endless recursion (that never completes)** before submitting to the Autograder. You can easily check for this by running in VSC and watching for program output, if it terminates timely or not.

4. Given that you already tried points 1-3, if you see that Autograder does not do anything (after you press 'submit') and waited for a while (30 seconds to 50 seconds), try refreshing the page or using a different browser.

5. Once you are done testing your code, comment out the tests i.e. the code under the __name__ == "__main__" section.

Problem 1: Root Finding with Newton-Raphson

We discussed in lecture the general problem of finding roots and how ubiquitous it is. Given a function f and interval $[a, b]$ find the $x' \in [a, b]$ (presuming it exists) such that $f(x') = 0$. We call x' a root. For example, if $f(x) = x^6 - x - 1$, $f(2) = 61$ and $f(1) = -1$. Then there must be some value $x' \in [1, 2]$ such that $f(x') = 0$. Using this approach we find

$$f(1.1347305283441975) = 6.573837356116385e - 05$$

Note that the number $6.573837356116385e - 05 = 0.00006573837356116385$ is so small that for the purpose of this problem, we can think of it as approximately equal to 0.

The Newton-Raphson is an algorithm to find roots that uses a function and its derivative to find a root. It is described recursively:

$$x_0 = estimate \tag{1}$$

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \tag{2}$$

To remind you, a derivative is a function that characterizes the way a function changes as inputs change. Equation 4 is the typical definition. Equation 5 is a common approximation of the derivative we saw in the last homework. Fig. 1 shows an iteration and the approximation of the root.

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h} \tag{3}$$

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h} \quad h \text{ is tiny, positive} \tag{4}$$

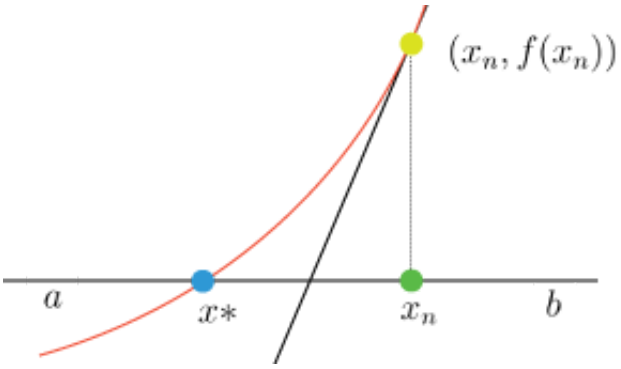

Figure 1: The root is $x*$. Our approximation x_n moves toward the root as long as we're larger than our threshold. Observe in the graphic that $f(b)$ is positive and $f(a)$ is negative insuring that there exists a root $x*$, $f(x*)$.

Another use of this approximation is to find maximal profit. Consider the following example:

$$\text{cost}(x) = 2000 + 500x \tag{5}$$

$$\text{revenue}(x) = 2000x - 10x^2 \tag{6}$$

$$\text{profit} = \text{revenue}(x) - \text{cost}(x) \tag{7}$$

You **must** observe that it is easy to write equation-7 in terms of x as: $(2000x - 10x^2) - (2000 + 500x)$. Since this is an equation, we can find the maximal profit by taking the derivative and solving for zero (finding the root). In this case, we'll find that the maximum profit occurs when $x = 75$ (however, we want to approximate the root rather than explicitly calculating the derivative and setting it to 0).

When the program is run we have the following output:

```
f(2) = 61
f(1) = -1
f(1.1347305283441975) = 6.573837356116385e-05 ~ 0.0
x = 74.99999941508389
The maximum profit is about $54250.0
```

with the plot:

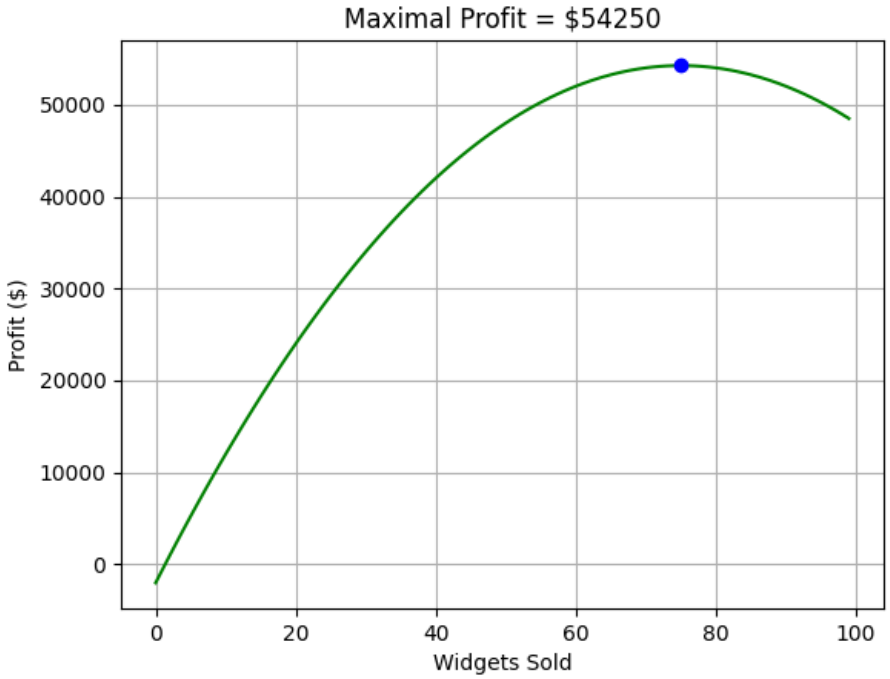

Figure 2: Using Newton-Raphson to find maximal profit $x = 75$.

Deliverables Programming Problem 1

- Complete the functions.
- You should use equation-4 for approximating the derivative.

Problem 2: Clustering with k-means, interpreting results, and visualization

Figure 3: *Versicolor* iris.

Clustering (or more formally, partitioning) is the most common AI task. We will be using scikit-learn's clustering implementation of the most commonly used algorith, k-means:

`https://scikit-learn.org/stable/modules/clustering.html`

Please read this page. In this problem, you'll cluster a collection of three kinds of irises, *Setosa*, *Versicolor*,*Virginica* based on four properties of the petal and sepal (protect the budding flower). So, in this problem, we have 150 flowers $F = \{f_0, \ldots, f_{149}\}$ each with four properties $f_i = (sl_i, sw_i, pl_i, pw_i)$ where sl is sepal length, sw is sepal width, pw is petal width, and pl is petal length. Using only these properties we want to cluster the flowers and create three blocks: $F = \{F_0, F_1, F_2\}$ where each of these blocks *should* contain only one type of iris. Because we actually know the species (not present in actual clustering), we can validate our results. The most common is the Rand Index. Please read about it here:

`https://scikit-learn.org/stable/modules/generated/sklearn.metrics.rand_score.html`

The data set F then contains 150 flowers with four properties and its actual species. We will show the task of clustering using k-means with $k = 3$ (number of blocks) on this data set and then you will implement a function to repeat the process but with only specific features (not all the features). The following code will help you to understand the steps of this process. When you implement the function, the code will laregely remain the same except few changes which we will explain after the code. Observe on line 11 that we're using all four features (or attributes).

```
1
2  #get data
3  iris = pd.read_csv('iris.csv')
4  print(iris.shape)
5  print(iris.head())
6
7  # prepare for clustering--use all the four properties
8  # leave out the actual species (we don't want to use the actual ↩
       species while clustering)
9  # Note the use iloc to subset the data and extract what we need
10 # Show correlation between all the features
11 iris_features = iris.iloc[:,[0,1,2,3]].values
12 corr = pd.DataFrame(iris.iloc[:,[0,1,2,3]]).corr()
13 print(corr)
14
15
16 # use scikit-learn and cluster the data in 3 blocks
17 result = KMeans(n_clusters=3,  random_state=0, n_init="auto")
18 iris_cluster_labels = result.fit_predict(iris_features)
19
20
21 # Print some information
```

```
print("The cluster labels:")
print(iris_cluster_labels)
print("The cluster centers:")
print(result.cluster_centers_)
rand_index = rand_score(iris_cluster_labels,iris['species'])
print(f"The rand index is {round(rand_index,2)}")



#show how pure each block is (meaning if it contains flowers of only one type or multiple types)
species = ['setosa', 'versicolor','virginica']
dsp = { j:[]  for j in species }
for i,j in enumerate(iris_cluster_labels):
    dsp[iris.species[i]].append(j)
print("The three clusters and counts of members")
for k,v in dsp.items():
    print(f"{k} {v.count(0),v.count(1),v.count(2)}")


#plot data colored by k-means labels with actual labels side-by-side
X,Y = [i[0] for i in iris_features],[i[1] for i in iris_features]
colors = [['b','g','c'][i] for i in iris_cluster_labels]
fig, ax = plt.subplots(1, 2)
sc1 = sn.scatterplot(data=iris,x='petal_width',y='petal_length',hue=iris_cluster_labels,ax=ax[0])
sc2 = sn.scatterplot(data=iris,x='petal_width',y='petal_length',hue='species',ax=ax[1])
sc1.set(title="K-means")
sc2.set(title="Actual")
plt.show()
```

has output

```
(150, 5)
   sepal_length  sepal_width  petal_length  petal_width species
0           5.1          3.5           1.4          0.2  setosa
1           4.9          3.0           1.4          0.2  setosa
2           4.7          3.2           1.3          0.2  setosa
3           4.6          3.1           1.5          0.2  setosa
4           5.0          3.6           1.4          0.2  setosa
              sepal_length  sepal_width  petal_length  petal_width
sepal_length      1.000000    -0.109369      0.871754     0.817954
sepal_width      -0.109369     1.000000     -0.420516    -0.356544
petal_length      0.871754    -0.420516      1.000000     0.962757
petal_width       0.817954    -0.356544      0.962757     1.000000
```

```
13 The cluster labels:
14 [1 1 1 1 1 1 1 1 1 1 1 1
15  1 1 1 1 1 1 1 1 1 1 1 1
16  1 1 1 1 1 1 1 1 1 1 1 1
17  1 1 1 1 1 1 1 1 1 1 2 0
18  2 0 0 0 0 0 0 0 0 0 0 0
19  0 0 0 0 0 0 0 0 0 0 0 2
20  0 0 0 0 0 0 0 0 0 0 0 0
21  0 0 0 0 0 0 0 0 0 2 0 2 2
22  2 2 0 2 2 2 2 2 2 0 0 2 2
23  2 2 0 2 0 2 0 2 2 0 0 2 2
24  2 2 2 0 2 2 2 2 0 2 2 2 0
25  2 2 2 0 2 2 0]
26 The cluster centers:
27 [[5.88360656 2.74098361 4.38852459 1.43442623]
28  [5.006      3.418      1.464      0.244     ]
29  [6.85384615 3.07692308 5.71538462 2.05384615]]
30 The rand index is 0.87
31 The three clusters and counts of members
32 setosa (0, 50, 0)
33 versicolor (47, 0, 3)
34 virginica (14, 0, 36)
```

The most important scikit-learn elements are lines 17-18. Line 17 creates a k-means object. We want three blocks. The remaining two parameters are standard (read for more information). Line 18 creates an array of so-called labels. These are integers 0,1,2 that represent the block to which each f belongs. The remainder of the code shows the elements of the clustering. The quality is the Rand Index 0.87 which is good. We use seaborn (that you must install, more information is given in the starter code) that is paired with pandas. Observe in Figure-4, we have colored the flowers by their respective class labels (obtained from Kmeans clustering) on the left, and by the actual species (flower type or actual label) on the right. As you can see, the clustering appears good.

In this problem, your task is to use the correlation matrix (lines 12-13) that indicate how pairs of columns are linearly correlated. The rule of thumb is that we want data that is **not** linearly correlated (least correlated are closest to 0). To that end, you'll complete a function lst_c_2(data) that takes the original data and returns three value:

1. The first is a 2 dimensional numpy array. The values of the columns of this array are the actual feature values of the least correlated columns.
2. The second is a tuple containing the actual indices of the least correlated columns.
3. The third is the absolute value of the correlation between the columns in the pair.

To make it simple, assume that our data looks as shown after the figure on line 1-5:

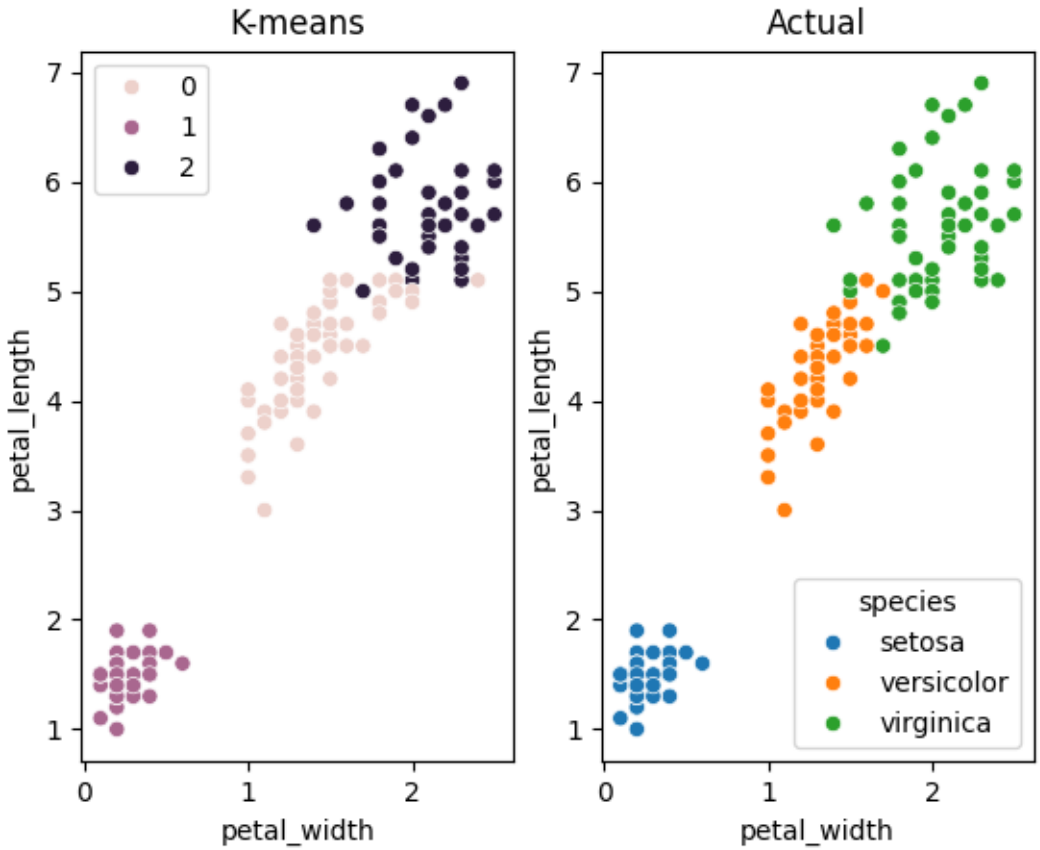

Figure 4: The left plot is k-means colored by class label. The right plot is the actual label using species. The separation between *Versicolor* and *Virginica* is the most difficult. Observe that the Rand Index is 0.87 indicating the cluster is good.

```
1    A    B    C    D
2    1    2    3    4
3    1    3    4    5
4    2    1    7    9
5    4    5    8    10
```

Assuming that least correlated columns are 'B' and 'D' at index 1 and 3 in the data, and the correlation between them is 0.74, then the output should be as follows:

```
1    [[2, 4], [3, 5], [1, 9], [5, 10]], (1, 3), 0.74
```

Please ensure that you return the indices in ascending order i.e. (2, 3) or (1, 2) not (3, 2) or (2, 1). The same is true for columns in the numpy array, if we find columsn 'B' and 'D' as least correlated then first column of numpy array should contain the values of 'B' and the second column should contain the values of 'D'.

The code is exactly the same as the original code that we explained before, except that you

will implement the lst_c_2():

```python
def lst_c_2(xpf):
    pass
...
features, pair_of_indices, value = lst_c_2(iris)
i,j = pair
print(f"Least corr columns: {iris.columns[i], iris.columns[j]} with {value}")
# using scikit-learn, cluster
...
```

producing

```
(150, 5)
   sepal_length  sepal_width  petal_length  petal_width species
0           5.1          3.5           1.4          0.2  setosa
1           4.9          3.0           1.4          0.2  setosa
2           4.7          3.2           1.3          0.2  setosa
3           4.6          3.1           1.5          0.2  setosa
4           5.0          3.6           1.4          0.2  setosa
Least corr columns: ('sepal_length', 'sepal_width') with 0.10936924995064931
The cluster labels:
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 2 1 2 1 2 1 2 2 2 2 2 2 1 2 2 2 2 2 2 2 2
 1 1 1 1 2 2 2 2 2 2 2 2 1 2 2 2 2 2 2 2 2 2 2 2 2 2 1 2 1 2 1 1 0 1 1 1 1
 1 1 2 2 1 1 1 1 2 1 2 1 2 1 1 2 2 1 1 1 1 1 2 2 1 1 1 2 1 1 1 2 1 1 1 2 1
 1 2]
The cluster centers:
[[5.00392157 3.4       ]
 [6.82391304 3.07826087]
 [5.8        2.7       ]]
The rand index is 0.82
The three clusters and counts of members:
setosa (50, 0, 0)
versicolor (0, 12, 38)
virginica (1, 34, 15)
```

and slightly different visualization (Figure-5), since you'll be clustering on only two of the features.

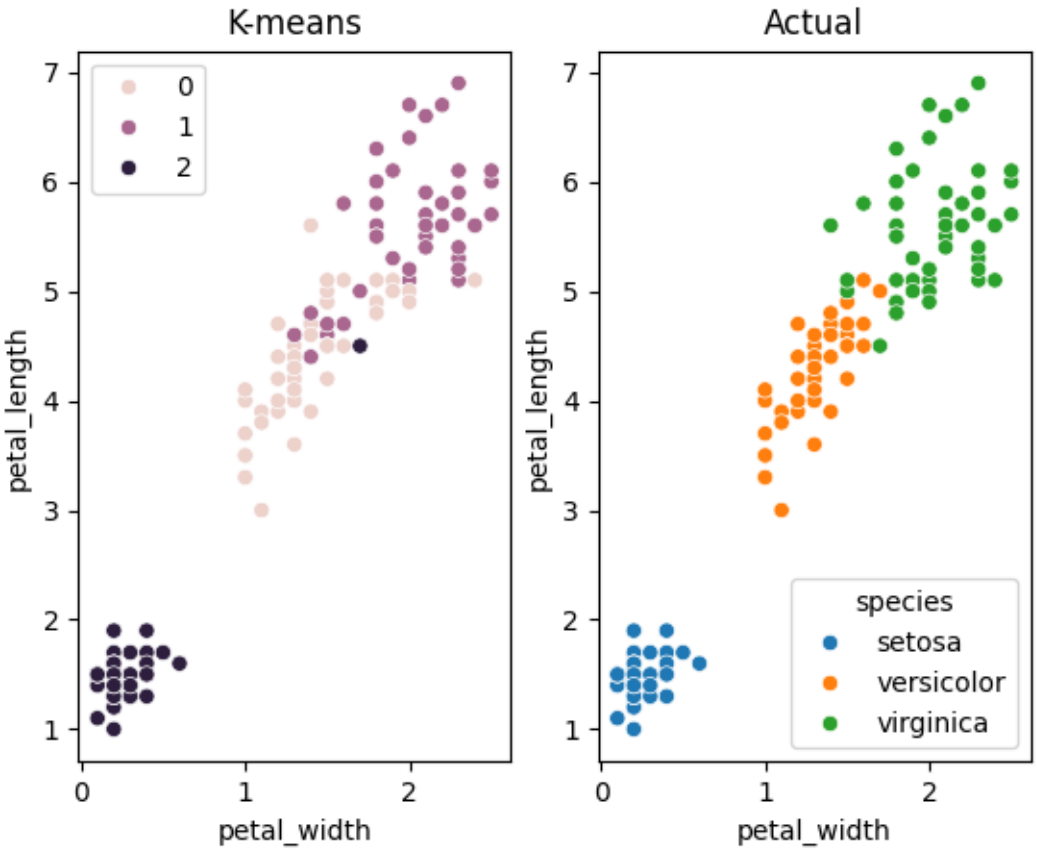

Figure 5: The left plot is k-means colored by class label using the two least correlated columns. The right plot is the actual label using species. The separation between *Versicolor* and *Virginica* is the most difficult. Observe that the Rand Index is 0.82 indicating the cluster is remains good, but a few of the flowers are further away from the block centers.

Deliverables Programming Problem 2

- You are provided with the original code to cluster under __main__.
- Read scikit-learn page on clustering to understand about input and output format and different parameters.
- Complete lst_c_2() that finds the least two correlated columns and returns an array containing values for those columns, a tuple of column IDs (as integers), and the absolute value of the correlation.

Problem 3: Simpson's Rule

In this problem, we will implement Simpson's Rule–a loop that approximates integration over an interval. Suppose we want to find the value of the integral below:

$$\int_a^b f(x)\,dx \tag{8}$$

We *could* use those pesky rules of integration–who's got time for all that, right? Or, as computer scientists, we could implement virtually all integration problems. Simpson's Rule is way of approximating an integration using parabolas (See Fig. 6). For the integration, we have to pick an even number of subintervals n and sum them up.

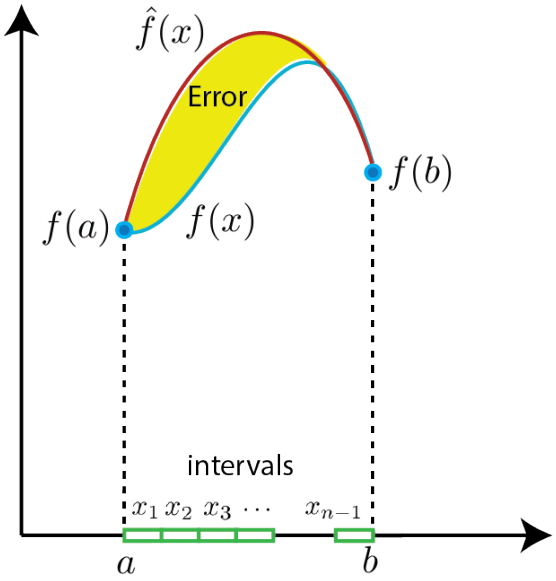

Figure 6: The function $f(x)$ integrated over a, b is approximated by $\hat{f}(x)$ using n equally sized invervals. The yellow illustrates the error of the approximation.

The *rule* is found on lines (13-14). Observe that when the index is odd that there is a coefficient of 4; when the index is even (excluding start and end, meaning $f(x0)$ and $f(xn)$ have no coefficients), the coefficient is 2.

$$\Delta x = \frac{b-a}{n} \tag{9}$$

$$x_i = a + i\Delta x, \qquad i = 0, 1, 2, \ldots, n-1, n \tag{10}$$

$$x_0 = a + 0\Delta x = a \tag{11}$$

$$x_n = a + n\frac{b-a}{n} = b \tag{12}$$

$$\int_a^b (x)\,dx \approx \frac{b-a}{3n}[f(x_0) + 4f(x_1) + 2f(x_2) + 4f(x_3) + 2f(x_4) + \ldots \tag{13}$$

$$+ 2f(x_{n-2} + 4f(x_{n-1}) + f(x_n)] \tag{14}$$

For example, the third row is the approximation to the integral

$$\int_0^\pi sin(x)\,dx = 2$$

using $n = 4$ intervals.

For those who want to verify,

$$\int_0^6 3t^2 + 1 = (\tfrac{1}{3})3t^3 + t\Big|_0^6 \tag{15}$$

$$= 216 + 6 = 222 \tag{16}$$

The following code:

```
data = [[lambda x:3*(x**2)+1, 0,6,2],[lambda x:x**2,0,5,6],
        [lambda x:math.sin(x), 0,math.pi, 4],[lambda x:1/x, 1, 11, 6]]

for d in data:
    f,a,b,n = d
    print(simpson(f,a,b,n))

area = simpson(lambda t: 3*(t**2) + 1,0,6,10)
t = np.arange(0.0, 10.0,.1)
fig,ax = plt.subplots()
s = np.arange(0,6.1,.1)
ax.plot(t, (lambda t: 3*(t**2) + 1)(t),'g')
plt.fill_between(s,(lambda t: 3*(t**2) + 1)(s))
ax.grid()
ax.set(xlabel ="x", ylabel=r"$f(x)=3x^2 + 1$",
title = r"Area under the curve $\int_0^6\,f(x)$ ~" + f"{round(area,2)}")
plt.show()
```

has output:

```
222.0
41.66666666666667
2.0045597549844207
2.4491973405016885
```

and plot as shown in Figure-7.

Programming Problem 3

- Complete the functions.

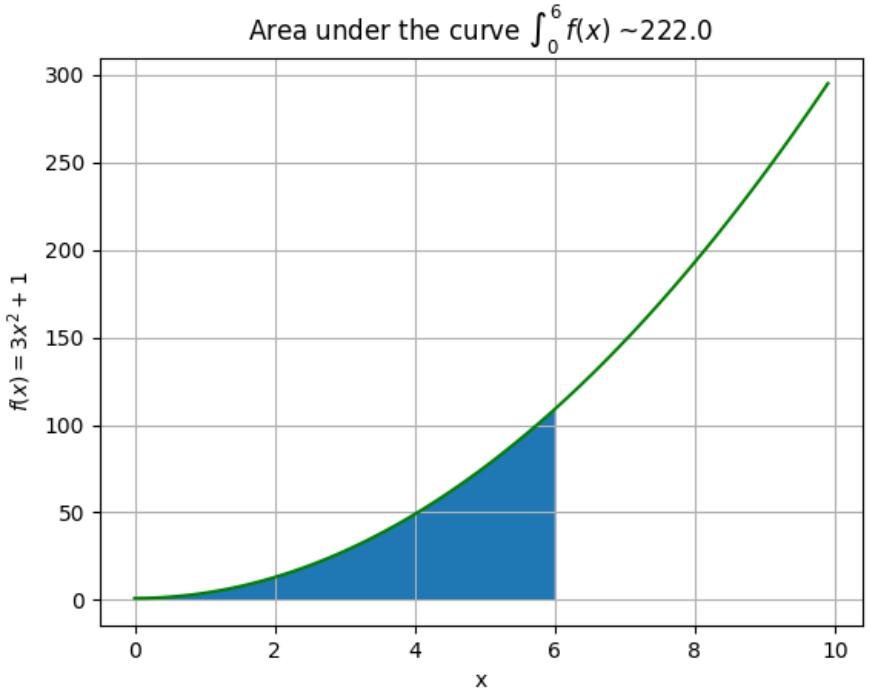

Figure 7: Using Simpson's rule to approximate an integral.

Problem 4: Central Dogma, DNA to RNA to Protein

The central dogma in biology is that DNA $\rightarrow$ RNA $\rightarrow$ protein. If you want, you can read more about it at `https://en.wikipedia.org/wiki/Central_dogma_of_molecular_biology`. In this problem you will read in a two files: The first file (amino_acids.txt) contains mapping of codons (a triplet of three DNA bases is called as **codon**) , by mapping we mean a rule that tells which specific codons will convert into a specific amino acid, and the second file (DNA.txt) contains a DNA sequence. The first file will help you to understand the mappings, and then by using the mappings, we will convert the sequence from the second file (DNA.txt) into amino acids. In essence, we will write a program to convert the DNA into protein as living organisms do! (a protein is a sequence of amino acids)

To start with, the contents of amino_acids.txt are shown below in the table.

Name	Abr.	Codons
Isoleucine	I	ATT, ATC, ATA
Leucine	L	CTT, CTC, CTA, CTG, TTA, TTG
Valine	V	GTT, GTC, GTA, GTG
Phenylalanine	F	TTT, TTC
Methionine	M	ATG
Cysteine	C	TGT, TGC
Alanine	A	GCT, GCC, GCA, GCG
Glycine	G	GGT, GGC, GGA, GGG
Proline	P	CCT, CCC, CCA, CCG
Threonine	T	ACT, ACC, ACA, ACG
Serine	S	TCT, TCC, TCA, TCG, AGT, AGC
Tyrosine	Y	TAT, TAC
Tryptophan	W	TGG
Glutamine	Q	CAA, CAG
Asparagine	N	AAT, AAC
Histidine	H	CAT, CAC
Glutamic acid	E	GAA, GAG
Aspartic acid	D	GAT, GAC
Lysine	K	AAA, AAG
Arginine	R	CGT, CGC, CGA, CGG, AGA, AGG
Stop codons	-	TAA, TAG, TGA

The first column is the full name of the amino acid. The second column is the abbreviation as one letter initial (for the same amino acid). For the stop codons, we use a dash. The rightmost column are what three letters of DNA are used to make the amino acid. The amino acid Arginine has an abbreviation R. There are six codon (three bases of DNA) that code for Arginine: CGT, CGC, CGA, CGG, AGA, AGG, as can be seen in the table.

About DNA.txt: It's basically a FASTA file. Please don't be confused by the name-it's just a text file that folllows certain rules, hence the special name. It has two parts: a header (information about the DNA sequence that it contains) and the DNA sequence itself. Here's the one you'll be using (don't copy them from here, we have already pushed both files to your repositories).

```
>HSGLTH1 Human theta 1-globin gene
CCACTGCACTCACCGCACCCGGCCAATTTTTGTGTT
TTTAGTAGAGACTAAATACCATATAGTGAACACCTA
AGACGGGGGGCCTTGGATCCAGGGCGATTCAGAGG
GCCCCGGTCGGAGCTGTCGGAGATTGAGCGCGCGC
GGTCCCGGGATCTCCGACGAGGCCCTGGACCCCCG
GGCGGCGAAGCTGCGGCGCGGCGCCCCCTGGAGGC
CGCGGGACCCCTGGCCGGTCCGCGCAGGCGCAGCG
GGGTCGCAGGGCGCGGCGGGTTCCAGCGCGGGGAT
GGCGCTGTCCGCGGAGGACCGGGCGCTGGTGCGCG
```

```
CCCTGTGGAAGAAGCTGGGCAGCAACGTCGGCGTCT
ACACGACAGAGGCCCTGGAAAGGTGCGGCAGGCTG
GGCGCCCCCGCCCCCAGGGGCCCTCCCTCCCCAAG
CCCCCCGGACGCGCCTCACCCACGTTCCTCTCGCAG
GACCTTCCTGGCTTTCCCCGCCACGAAGACCTACTT
CTCCCACCTGGACCTGAGCCCCGGCTCCTCACAAGT
CAGAGCCCACGGCCAGAAGGTGGCGGACGCGCTGA
GCCTCGCCGTGGAGCGCCTGGACGACCTACCCCAC
GCGCTGTCCGCGCTGAGCCACCTGCACGCGTGCCA
GCTGCGAGTGGACCCGGCCAGCTTCCAGGTGAGCG
GCTGCCGTGCTGGGCCCCTGTCCCCGGGAGGGCCC
CGGCGGGGTGGGTGCGGGGGGCGTGCGGGGCGGG
TGCAGGCGAGTGAGCCTTGAGCGCTCGCCGCAGCT
CCTGGGCCACTGCCTGCTGGTAACCCTCGCCCGGCA
CTACCCCGGAGACTTCAGCCCCGCGCTGCAGGCGTC
GCTGGACAAGTTCCTGAGCCACGTTATCTCGGCGCT
GGTTTCCGAGTACCGCTGAACTGTGGGTGGGTGGCC
GCGGGATCCCCAGGCGACCTTCCCCGTGTTTGAGTA
AAGCCTCTCCCAGGAGCAGCCTTCTTGCCGTGCTCT
CTCGAGGTCAGGACGCGAGAGGAAGGCGC
```

Though not necessary but if you want, you can read about this gene here: `https://pubmed.ncbi.nlm.nih.gov/3422341/`. In the file, the first line describes the sequence providing the name and other attributes. The remaining lines in the FASTA file is the DNA sequence, where all lines after the header are part of the same sequence so there are no line breaks (ignore all white space).

Example: how to translate DNA into a protein

To convert from DNA to protein, we use a sequence of codons. We'll bold the protein when its translated.

Let's look at the first twelve bases: CCACTGCACTCA. Every three bases uniquely determine an amino acid.

1. Start with the first codon CCA, CCACTGCACTCA.
2. Looking at the first file we see: Proline, P, CCT, CCC, CCA, CCG. This means we can rewrite CCA as P.
3. Looking at the next codon CTG, **P**CTGCACTCA
4. We find it matches Leucine, L, CTT, CTC, CTA, CTG, TTA, TTG. So our protein is PL.
5. The next three are CAC **PL**CACTCA.
6. The table has Histidine, H, CAT, CAC.

7. The final three are TCA. **PLH**TCA

8. This matches Serine, S, TCT, TCC, TCA, TCG, AGT, AGC.

9. The protein is **PLHS**.

If you are at the end and only have two bases, you cannot match a protein, so you ignore them. Suppose we had CCAC. We know CCA is P. Then we only have C left. We ignore it.

How to approach this problem

Our main task is to take the DNA (by reading DNA.txt) and produce a string of single letters (as shown in above example, points 1-9) that reflect the encoding. Here is one way to solve this (you should implement how you feel most comfortably)

1. First you'll read in the table from amino_acids.txt, and create a dictionary whose entries are:

$$aa\_d \quad = \quad \{(c_0, c_1, \ldots, c_n) : [name, letter], \ldots\}$$

where c_i is a three letter codon, $name$ is the full name of the amino acid, and $letter$ is the single letter for the amino acid. Make sure you follow the format of keys and values exactly as shown here. The function get_amino_acid() takes the file name and returns a dictionary. The dictionary is also shown below in the code listing.

2. Read in the DNA sequence, the function get_DNA() takes a file name and returns a faste data structure [header, DNA] (FASTA data structure) where header is the first line of the file DNA.txt and DNA is the DNA sequence (the sequence of A,T,G,C after the first line) (ignoring any whitespace). The read-in FASTA sequence is also shown below in the code listing.

3. The function translate() takes a FASTA data structure (that you created in step-2) and returns a string that is the translation using the dictionary (that you created in step-1). We can do a simple print to see whether our translation is the same as actual. An example of translate() function is also shown below in the code listing (you are encouraged to cross-check via pen and paper).

This code creates the dictionary and FASTA file (as a list), translates, and validates:

```
print("Dictionary")
print(aa_d)
print("FASTA file")
print(DNA_d)
print("Translations match:", str(protein == actual))

#should return "PLHS"
print(translate(["nothing", "CCACTGCACTCA"]))
```

```
9
10 #should return "D-"
11 print(translate(["nothing", "GACTAA"]))
```

has output:

```
1  Dictionary
2  {('ATT', 'ATC', 'ATA'): ['Isoleucine', 'I'], ('CTT', 'CTC', 'CTA', '↩
       CTG', 'TTA', 'TTG'): ['Leucine', 'L'], ('GTT', 'GTC', 'GTA', 'GTG')↩
       : ['Valine', 'V'], ('TTT', 'TTC'): ['Phenylalanine', 'F'], ('ATG',)↩
       : ['Methionine', 'M'], ('TGT', 'TGC'): ['CYSteine', 'C'], ('GCT', '↩
       GCC', 'GCA', 'GCG'): ['Alanine', 'A'], ('GGT', 'GGC', 'GGA', 'GGG')↩
       : ['Glycine', 'G'], ('CCT', 'CCC', 'CCA', 'CCG'): ['Proline', 'P'],↩
        ('ACT', 'ACC', 'ACA', 'ACG'): ['Threonine', 'T'], ('TCT', 'TCC', '↩
       TCA', 'TCG', 'AGT', 'AGC'): ['Serine', 'S'], ('TAT', 'TAC'): ['↩
       Tyrosine', 'Y'], ('TGG',): ['Tryptophan', 'W'], ('CAA', 'CAG'): ['↩
       Glutamine', 'Q'], ('AAT', 'AAC'): ['Asparagine', 'N'], ('CAT', 'CAC↩
       '): ['Histidine', 'H'], ('GAA', 'GAG'): ['Glutamic_acid', 'E'], ('↩
       GAT', 'GAC'): ['AsparTic acid', 'D'], ('AAA', 'AAG'): ['Lysine', 'K↩
       '], ('CGT', 'CGC', 'CGA', 'CGG', 'AGA', 'AGG'): ['Arginine', 'R'], ↩
       ('TAA', 'TAG', 'TGA'): ['Stop_codons', '-']}
3  FASTA file
4  ['>HSGLTH1 Human theta 1-globin gene', '↩
       CCACTGCACTCACCGCACCCGGCCAATTTTTGTGTTTTTAGT
5  AGAGACTAAATACCATATAGTGAACACCTAAGACGGGGGGC
6  CTTGGATCCAGGGCGATTCAGAGGGCCCCGGTCGGAGCTGT
7  CGGAGATTGAGCGCGCGCGGTCCCGGGATCTCCGACGAGGC
8  CCTGGACCCCCGGGCGGCGAAGCTGCGGCGCGGCGCCCCCT
9  GGAGGCCGCGGGACCCCTGGCCGGTCCGCGCAGGCGCAGCG
10 GGGTCGCAGGGCGCGGCGGGTTCCAGCGCGGGGATGGCGCT
11 GTCCGCGGAGGACCGGGCGCTGGTGCGCGCCCTGTGGAAGA
12 AGCTGGGCAGCAACGTCGGCGTCTACACGACAGAGGCCCTG
13 GAAAGGTGCGGCAGGCTGGGCGCCCCCGCCCCCAGGGGCCC
14 TCCCTCCCCAAGCCCCCCGGACGCGCCTCACCCACGTTCCTC
15 TCGCAGGACCTTCCTGGCTTTCCCCGCCACGAAGACCTACTT
16 CTCCCACCTGGACCTGAGCCCCGGCTCCTCACAAGTCAGAGC
17 CCACGGCCAGAAGGTGGCGGACGCGCTGAGCCTCGCCGTGG
18 AGCGCCTGGACGACCTACCCCACGCGCTGTCCGCGCTGAGC
19 CACCTGCACGCGTGCCAGCTGCGAGTGGACCCGGCCAGCTT
20 CCAGGTGAGCGGCTGCCGTGCTGGGCCCCTGTCCCCGGGAG
21 GGCCCCGGCGGGGTGGGTGCGGGGGGCGTGCGGGGCGGGT
22 GCAGGCGAGTGAGCCTTGAGCGCTCGCCGCAGCTCCTGGGC
23 CACTGCCTGCTGGTAACCCTCGCCCGGCACTACCCCGGAGAC
24 TTCAGCCCCGCGCTGCAGGCGTCGCTGGACAAGTTCCTGAGC
25 CACGTTATCTCGGCGCTGGTTTCCGAGTACCGCTGAACTGTG
26 GGTGGGTGGCCGCGGGATCCCCAGGCGACCTTCCCCGTGTTTG
```

```
27 AGTAAAGCCTCTCCCAGGAGCAGCCTTCTTGCCGTGCTCTCTC
28 GAGGTCAGGACGCGAGAGGAAGGCGC']
29 Translations match: True
30 PLHS
31 D-
```

Deliverables Problem 4

- Carefully read the instructions in the starter file a8.py. You may not be able to run it directly in VSC, so follow the instructions in the starter code.
- Complete the functions get_DNA(), get_amino_acids() and translate() as per the specifications.
- You are allowed to use replace for space and '{'. Other uses of it will actually be more difficult.
- For this problem, using pandas makes the autograder a little crazy–so we're eschewing dataframes. Furthermore, everything is text.

Problem 5: Marginal Cost

When producing something, the cost typically varies. The **marginal cost** is the cost incurred by producing an additonal unit of product or service. This is the derivative in disguise. Given a function $cost(x)$, we can determine the derivative as:

$$\frac{d\,cost(s)}{dx} \approx \lambda x : \frac{cost(x+h) - cost(x-h)}{2h} \tag{17}$$

for a very small value of h. For this problem assume the cost function is:

$$cost(x) = 0.0001x^3 - 0.08x^2 + 40x + 5000 \tag{18}$$

The following code:

```
U,C = [],[]
for unit in range(200,650,50):
    U.append(unit)
    mc = round(marginal_cost(cost)(unit),0)
    C.append(mc)
    print(f"For {unit} marginal cost is {mc}")
plt.plot(U,C,'b-')
plt.plot(300,round(marginal_cost(cost)(300)),'ro')
plt.xlabel("Units of Production")
plt.ylabel("Cost $")
plt.title(r"Marginal cost Cost(x) =  $0.0001x^3 - 0.08x^2 + 40x + 5000$")
plt.show()
```

has the output:

```
For 200 marginal cost is 20.0
For 250 marginal cost is 19.0
For 300 marginal cost is 19.0
For 350 marginal cost is 21.0
For 400 marginal cost is 24.0
For 450 marginal cost is 29.0
For 500 marginal cost is 35.0
For 550 marginal cost is 43.0
For 600 marginal cost is 52.0
```

and plot:

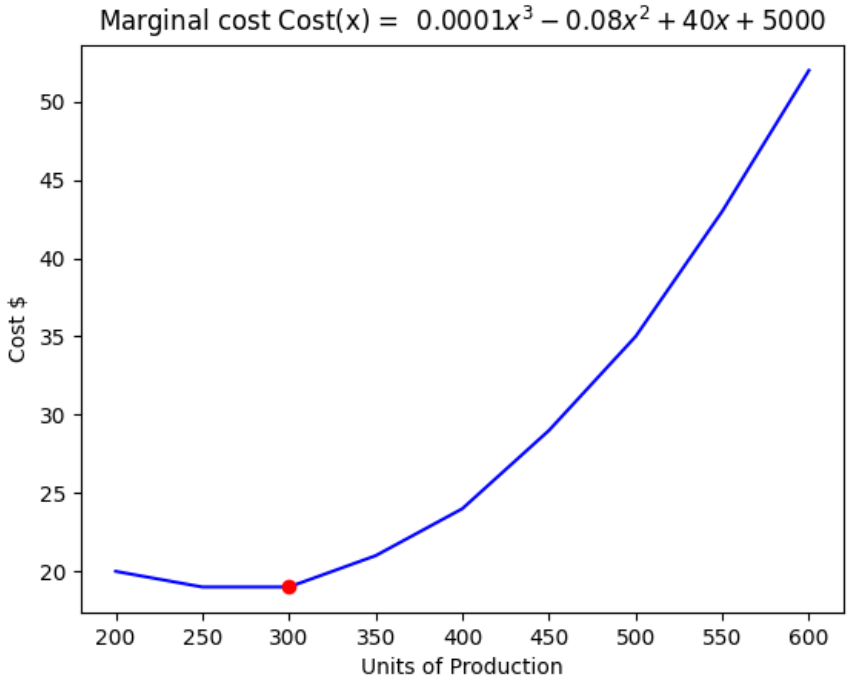

Figure 8: Marginal cost for sales 200-600 units. Observe the cost increases at 300.

Deliverables Problem 5

- We include the cost function for you.
- This is a very short homework problem, since you've already implemented the derivative function earlier!

Extra Credit: Polynomial Interpolation

This problem is only counted as extra credit. To that end, it can involve more complexity and should only be attempted if the other problems are completed. Most curves we encounter can be effectively modeled by polynomials. We've seen numpy's approach (we'll talk about it more in a subsequent homework), but there exists an older technique: Newton's Binomial Interpolation Formula. Understanding the theory isn't necessary to understand the formula, but it does include an extension of the binomial formula $\binom{i}{j}$ for $i, j = 0, 1, 2, \ldots$. In this case the expression:

$$\binom{x}{0} = \tfrac{1}{0!}1 \tag{19}$$

$$\binom{x}{1} = \tfrac{1}{1!}x \tag{20}$$

$$\binom{x}{2} = \tfrac{1}{2!}(x-1)x \tag{21}$$

$$\binom{x}{3} = \tfrac{1}{3!}(x-2)(x-1)x \tag{22}$$

$$\binom{x}{n} = \tfrac{1}{n!}(x-(n-1))(x-(n-2))\cdots(x-1)x \tag{23}$$

for a variable x and integer n. The approach uses a difference table which we'll discuss next. An difference operator Δ, for a list of values $\ell = [y_0, y_1, \ldots, y_n]$ produces a new list $\Delta(\ell) = [y_1 - y_0, y_2 - y_1, \ldots, y_n - y_{n-1}]$. If we write the exponent as the number of times Δ is applied, we have a difference table is:

$$dtable(\ell) = [\Delta^0(\ell), \Delta^1(\ell), \Delta^2(\ell), \ldots, \Delta^i(\ell)], i = \text{len}(\ell) - 1 \tag{24}$$

for list $\text{len}(\ell) > 1$. Assume $\ell = [-1, 0, 5, 20]$. The table can be visualized as:

ℓ	$\Delta(\ell)$	$\Delta^2(\ell)$	$\Delta^3(\ell)$
-1			
	1		
0		4	
	5		6
5		10	
	15		
20			

The code confirms this:

```
1 lst = [-1,0,5,20]
2 print(dtable(lst)) #function that takes a list and returns a ↩
    difference table
```

produces:

```
[[-1, 0, 5, 20], [1, 5, 15], [4, 10], [6]]
```

We need the first value in each list. Let $\rho = [\ell[0], \Delta(\ell)[0], \Delta^2(\ell)[0], \ldots, \Delta^i(\ell)[0]]$. For this example,

$$\rho = [-1, 1, 4, 6]$$

. An interesting relationship arises:

$$\ell[-1] = \rho_0 + \binom{i}{1}\rho_1 + \binom{i}{2}\rho_2 + \cdots + \binom{i}{i}\rho_i \tag{25}$$

where $i = \text{len}(\ell) - 1$. In our example,

$$20 = \binom{3}{0}(-1) + \binom{3}{1}1 + \binom{3}{2}4 + \binom{3}{3}6 \tag{26}$$

$$= -1(1) + 3(1) + 3(4) + 6 = 20 \tag{27}$$

The code:

```
lst = [-1,0,5,20]

print(rho(lst))

i = len(lst) - 1
s_ = [math.comb(i,j)*r_ for j,r_ in enumerate(rho(lst))]
print(lst[-1], sum(s_))
```

produces

```
[-1, 1, 4, 6]
20 20
```

Let $D = \{(x_0, y_0), (x_1, y_1), \ldots, (x_n, y_n)\}$. Then we can build a polynomial that describes D:

$$I(D) = \binom{x}{0}\rho_0 + \binom{x}{1}\rho_1 + \binom{x}{2}\rho_2 + \cdots + \binom{x}{i}\rho_i \tag{28}$$

where $i = n - 1$ and ρ_i are values from the difference table generated from the y values as before. For example, let $D = [(0, -2), (1, 5), (2, 7), (3, 10)]$. Then:

$$I(D) = -2 + 7\binom{x}{1} - 5\binom{x}{2} + 6\binom{x}{3} \tag{29}$$

$$= -2 + 7x - \tfrac{5}{2}x(x-1) + x(x-1)(x-2) \tag{30}$$

$$= x^3 - \tfrac{11}{2}x^2 + \tfrac{23}{2}x - 2 \tag{31}$$

We don't need to simplify, but do it to write a simple function to plot. The following code:

```
D = [(0,-2),(1,5),(2,7),(3,10)]
```

```
2  pf = lambda x:x**3 - (11/2)*(x**2) + (23/2)*x - 2
3  f = build_polyonomial(D)
4  for i in range(7):
5      print(i,pf(i),f(i))
6  x = np.linspace(0,6,100)
7  X,Y = [x for x,_ in D],[y for _,y in D]
8  plt.plot(X,Y,'ro')
9  plt.plot(x,f(x),'b-')
10 plt.title("Newton Binomial Interpolation Formula")
11 plt.show()
```

produces:

```
1 0 -2.0 -2.0
2 1 5.0 5.0
3 2 7.0 7.0
4 3 10.0 10.0
5 4 20.0 20.0
6 5 43.0 43.0
7 6 85.0 85.0
```

and plot shown in Figure-9.

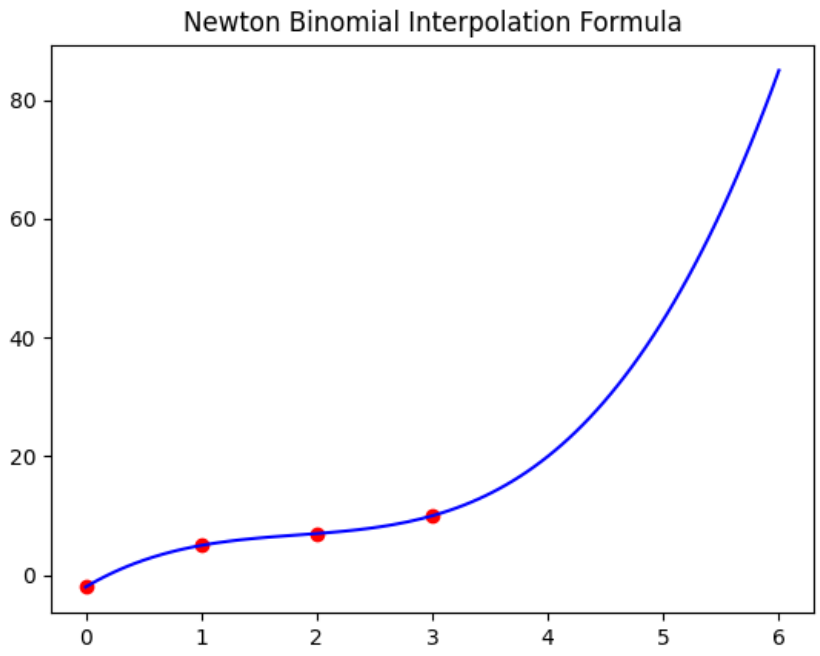

Figure 9: The points points are the data D = [(0,-2),(1,5),(2,7),(3,10)]. The blue curve the fitted polynomial.

Line 3 sends the data to a function build_polynomial that returns the interpolated polynomial

as a lambda function. Observe this function is the same as Eq. (31).

Deliverables Extra Credit

- Complete the build_polynomial function
- There are many approaches to building the polynomial (once the difference table is built). One approach is to treat the functional binomial $\binom{x}{n}$ as a λ function. You'll likely have to build a function (as a string) and then use eval to make it useable.
- All functions used in this problem are local to build_polynomial.
- If you build a test case, use $D = [(0, y_0), (1, y_1), \ldots, (k, y_k)])$ where successive x values differ by only 1 and start at zero. This is done to simplify the algorithm. These can be changed in a more general solution.

Programming pairs

lflenoy@iu.edu, tangtom@iu.edu
makinap@iu.edu, mszczas@iu.edu
tchapell@iu.edu, asultano@iu.edu
ovadeley@iu.edu, gsilingh@iu.edu
aberkun@iu.edu, ntorpoco@iu.edu
cuizek@iu.edu, etprince@iu.edu
nfarhat@iu.edu, patelsak@iu.edu
gcopus@iu.edu, mjroelle@iu.edu
skatiyar@iu.edu, annaum@iu.edu
ethickma@iu.edu, mmunaf@iu.edu
lpfritsc@iu.edu, rorymurp@iu.edu
brownset@iu.edu, ragmahaj@iu.edu
kaihara@iu.edu, sahimann@iu.edu
maladwa@iu.edu, ruska@iu.edu
schinitz@iu.edu, sasayini@iu.edu
cfampo@iu.edu, dyashwar@iu.edu
jwcase@iu.edu, jwu6@iu.edu
ag69@iu.edu, mveltri@iu.edu
sydecook@iu.edu, savebhat@iu.edu
masharre@iu.edu, ysanghi@iu.edu
milhavil@iu.edu, wwtang@iu.edu
egoldsto@iu.edu, gavilleg@iu.edu
nharkins@iu.edu, btasa@iu.edu
jacklapp@iu.edu, fshamrin@iu.edu
howamatt@iu.edu, aledminc@iu.edu

hh35@iu.edu, asteini@iu.edu
cpkerns@iu.edu, apathma@iu.edu
ahavlin@iu.edu, asaokho@iu.edu
aakindel@iu.edu, nsatti@iu.edu
coopjose@iu.edu, abiparri@iu.edu
spgreenf@iu.edu, hasiddiq@iu.edu
cannan@iu.edu, jneblett@iu.edu
adhuria@iu.edu, vrradia@iu.edu
howardbw@iu.edu, bmpool@iu.edu
grafe@iu.edu, adsize@iu.edu
jakchap@iu.edu, arirowe@iu.edu
apbabu@iu.edu, iperine@iu.edu
stkimani@iu.edu, rwan@iu.edu
arnadutt@iu.edu, ijvelmur@iu.edu
kjj6@iu.edu, wlyzun@iu.edu
lawmat@iu.edu, rnschroe@iu.edu
ejhaas@iu.edu, asidda@iu.edu
fkeele@iu.edu, rtrammel@iu.edu
kraus@iu.edu, aveluru@iu.edu
evacoll@iu.edu, fmahamat@iu.edu
tychid@iu.edu, kamaharj@iu.edu
hk120@iu.edu, pw18@iu.edu
krisgupt@iu.edu, lmeldgin@iu.edu
nfelici@iu.edu, reddyrr@iu.edu
colrkram@iu.edu, dsummit@iu.edu
dja1@iu.edu, brayrump@iu.edu
keswar@iu.edu, ir1@iu.edu
rl29@iu.edu, ansakrah@iu.edu
daxbills@iu.edu, apavlako@iu.edu
saecohen@iu.edu, justyou@iu.edu
alscarr@iu.edu, emisimps@iu.edu
kekchoe@iu.edu, mehtriya@iu.edu
ckdiallo@iu.edu, benprohm@iu.edu
micahand@iu.edu, pp31@iu.edu
mohiambu@iu.edu, smremmer@iu.edu
tcconnol@iu.edu, maklsmit@iu.edu
ryanbren@iu.edu, lmadiraj@iu.edu
ajgrego@iu.edu, snyderjk@iu.edu
dblackme@iu.edu, schwajaw@iu.edu
ohostet@iu.edu, dukthang@iu.edu
jabbarke@iu.edu, blswing@iu.edu

twfine@iu.edu, rpoludas@iu.edu
rcaswel@iu.edu, patel89@iu.edu
mdiazrey@iu.edu, patekek@iu.edu
bjdahl@iu.edu, masmatth@iu.edu
bcdutka@iu.edu, reedkier@iu.edu
mrcoons@iu.edu, voram@iu.edu
daminteh@iu.edu, jcn1@iu.edu
oakinsey@iu.edu, jdw14@iu.edu
mbeigie@iu.edu, kvpriede@iu.edu
kapgupta@iu.edu, aselki@iu.edu
gmhowell@iu.edu, gs29@iu.edu
wilcusic@iu.edu, krbpatel@iu.edu
ethbrock@iu.edu, jaslnu@iu.edu
abellah@iu.edu, vpolu@iu.edu
earuland@iu.edu, drsnid@iu.edu
kaneai@iu.edu, skp2@iu.edu
loggreen@iu.edu, megapaul@iu.edu
davgourl@iu.edu, ammulc@iu.edu
cgkabedi@iu.edu, wilsori@iu.edu
edfran@iu.edu, jpochyly@iu.edu
ameydesh@iu.edu, pricemo@iu.edu
adwadash@iu.edu, jtsuter@iu.edu
blacount@iu.edu, qshamsid@iu.edu
apchavis@iu.edu, mmarotti@iu.edu
mdonato@iu.edu, lvansyck@iu.edu
simadams@iu.edu, sahaan@iu.edu
laharden@iu.edu, lpelaez@iu.edu
zacbutle@iu.edu, jomeaghe@iu.edu
mdoxsee@iu.edu, ntuhl@iu.edu
fkanmogn@iu.edu, jsadiq@iu.edu
bencho@iu.edu, cstancom@iu.edu
wgurley@iu.edu, emluplet@iu.edu
dce@iu.edu, jwember@iu.edu
leokurtz@iu.edu, wardjohn@iu.edu
escolber@iu.edu, scotbray@iu.edu
aketcha@iu.edu, pateishi@iu.edu
nbernot@iu.edu, evataylo@iu.edu
ceub@iu.edu, ajtse@iu.edu
gandhira@iu.edu, wtrucker@iu.edu
arklonow@iu.edu, cltran@iu.edu
mbrockey@iu.edu, isaramir@iu.edu

anrkram@iu.edu, nrizvi@iu.edu
ek37@iu.edu, woodsky@iu.edu
aaamoako@iu.edu, emgward@iu.edu
leegain@iu.edu, lukastef@iu.edu
aaragga@iu.edu, leolin@iu.edu
mkames@iu.edu, owysmit@iu.edu
allencla@iu.edu, jarlmint@iu.edu
wanjiang@iu.edu, jnzheng@iu.edu
skunduru@iu.edu, patedev@iu.edu
maudomin@iu.edu, cmarcuka@iu.edu
maxklei@iu.edu, aidnschi@iu.edu
ajeeju@iu.edu, anajmal@iu.edu
nmcastan@iu.edu, jactrayl@iu.edu
bellcol@iu.edu, rvinzant@iu.edu
spgerst@iu.edu, nniranj@iu.edu
bkante@iu.edu, avraya@iu.edu
ruqchen@iu.edu, rosenbbj@iu.edu
austdeck@iu.edu, tolatinw@iu.edu
jhar@iu.edu, myeralli@iu.edu
jtbland@iu.edu, keasandl@iu.edu
bencalex@iu.edu, wtubbs@iu.edu
ridbhan@iu.edu, jlzhao@iu.edu
matgarey@iu.edu, bdyiga@iu.edu
seangarc@iu.edu, chrimanu@iu.edu
joehawl@iu.edu, madymcsh@iu.edu
jdc6@iu.edu, cnyarko@iu.edu
lcoveney@iu.edu, awsaunde@iu.edu
quecox@iu.edu, erschaef@iu.edu
ayoajayi@iu.edu, audtravi@iu.edu
jkielcz@iu.edu, gmpierce@iu.edu
liansia@iu.edu, aditpate@iu.edu
marganey@iu.edu, surapapp@iu.edu
spdamani@iu.edu, ism1@iu.edu
oeichenb@iu.edu, aamathew@iu.edu
zguising@iu.edu, mzagotta@iu.edu
hermbrar@iu.edu, ryarram@iu.edu
mkleinke@iu.edu, lqadan@iu.edu
diebarro@iu.edu, vyeruba@iu.edu
sfuneno@iu.edu, clmcevil@iu.edu
josespos@iu.edu, antando@iu.edu
khannni@iu.edu, sezinnkr@iu.edu

nihanas@iu.edu, linjaso@iu.edu
agawrys@iu.edu, giomayo@iu.edu
avulas@iu.edu, sahishah@iu.edu
amkhatri@iu.edu, joshroc@iu.edu
flynncj@iu.edu, amanocha@iu.edu
anlego@iu.edu, jwmullis@iu.edu
fu7@iu.edu, ap79@iu.edu
althart@iu.edu, gavsteve@iu.edu
alchatz@iu.edu, rafir@iu.edu
greenpat@iu.edu, muyusuf@iu.edu
brhint@iu.edu, thnewm@iu.edu
nokebark@iu.edu, utwade@iu.edu
garcied@iu.edu, tzuyyen@iu.edu
johnguen@iu.edu, vmungara@iu.edu
hamac@iu.edu, aptheria@iu.edu
mrfehr@iu.edu, ksadiq@iu.edu
jc168@iu.edu, gepearcy@iu.edu
jwdrew@iu.edu, mz24@iu.edu
jhhudgin@iu.edu, crmoll@iu.edu
efritch@iu.edu, samrile@iu.edu
agrevel@iu.edu, samyuan@iu.edu
adiyer@iu.edu, cmvanhov@iu.edu
hawkjod@iu.edu, nmr1@iu.edu
kdembla@iu.edu, epautsch@iu.edu
huhasan@iu.edu, bcmarret@iu.edu
delkumar@iu.edu, ltmckinn@iu.edu
vkommar@iu.edu, gszopin@iu.edu
phjhess@iu.edu, jwetherb@iu.edu
nolakim@iu.edu, nlippman@iu.edu
caegrah@iu.edu, aranjit@iu.edu
dkkosim@iu.edu, tpandey@iu.edu
clearle@iu.edu, nichojop@iu.edu
mwclawso@iu.edu, zshamo@iu.edu
saganna@iu.edu, coenthom@iu.edu
eakanle@iu.edu, cjwaller@iu.edu
jacobben@iu.edu, pravulap@iu.edu
coopelki@iu.edu, tarturnm@iu.edu
achordi@iu.edu, dwo@iu.edu
tfreson@iu.edu, majtorm@iu.edu
sg40@iu.edu, ao9@iu.edu
jdemirci@iu.edu, rt11@iu.edu

sakalwa@iu.edu, impofujr@iu.edu
sgaladim@iu.edu, anemlunc@iu.edu
aroraarn@iu.edu, deturne@iu.edu
alelefeb@iu.edu, liwitte@iu.edu
fdonfrio@iu.edu, orrostew@iu.edu
swconley@iu.edu, mnimmala@iu.edu
ddrotts@iu.edu, clscheum@iu.edu
jdgonzal@iu.edu, dernguye@iu.edu
migriswo@iu.edu, cialugo@iu.edu
deombeas@iu.edu, wtatoole@iu.edu
laburkle@iu.edu, thomps16@iu.edu