**ENGR 102 Sect 508 Lab 12a**

**100 points+[55points]**

**Reading assignment:**

|  |  |
| --- | --- |
| **Lecture Slides** | **L12** |
| **zyBook chapter 12** | **Complete all participation and challenge activities** |

*Attention!!*

*Team submission. one submission per team.*

*Submit* *your Py-files together with your word/pdf file with screenshots of your tests outputs. Include any derivations, comments and supplemental notes in your word/pdf files.*

*No pictures by the phone – it is impossible to read. You will be allowed to resubmit and reupload HW as many times as you want to within the due date/time, only last submission will be graded. No late submissions. For submission you may use this file as template: rename file including your name. Do not forget to put your name inside of this file as well. If it is a team work use Team Header, include the team number and all team members.*

**[60 points] Activity #1: Plotting Curves and Derivatives – to be done as a team**

As a team, you are to develop a program that will ask a user for a polynomial (of arbitrary degree) and then will generate a plot of the curve, its derivative curve, and its second derivative curve. All three should be plotted on the same axes so that it is easy to see how the first and second derivatives affect the shape of the curve. You are also to highlight local maxima and minima by drawing points on the plots showing where these occur. You may use code developed in prior weeks if it is helpful (but it will probably be as easy, at this point, to write new code).

* The user should be able to enter the polynomial as a set of coefficients. You will need to be sure to clearly tell the user the format to enter these coefficients in.
* You should determine the derivatives analytically. That is, if you have a list of the coefficients of the original polynomial, you should be able to generate the first and second derivatives exactly.
* The plots can be over a predetermined range of x and y values, but should be sufficiently large that someone entering a polynomial would easily be able to generate polynomials passing through the area covered by the plots. Your plots can be generated by sampling many points on the curves and plotting a line connecting those points.
* You should identify the local maxima and minima in the range you are plotting. You can do this by looking through the points generated for your plot, and finding ones that are local maxima (a point is greater than those just before and after) and local minima (a point is smaller than those just before and after). These points should be plotted on top of the plots of the curves.
* Your plots for the three curves and the maxima/minima should be sufficiently varied (in color/shape) that it is easy to distinguish one from another.

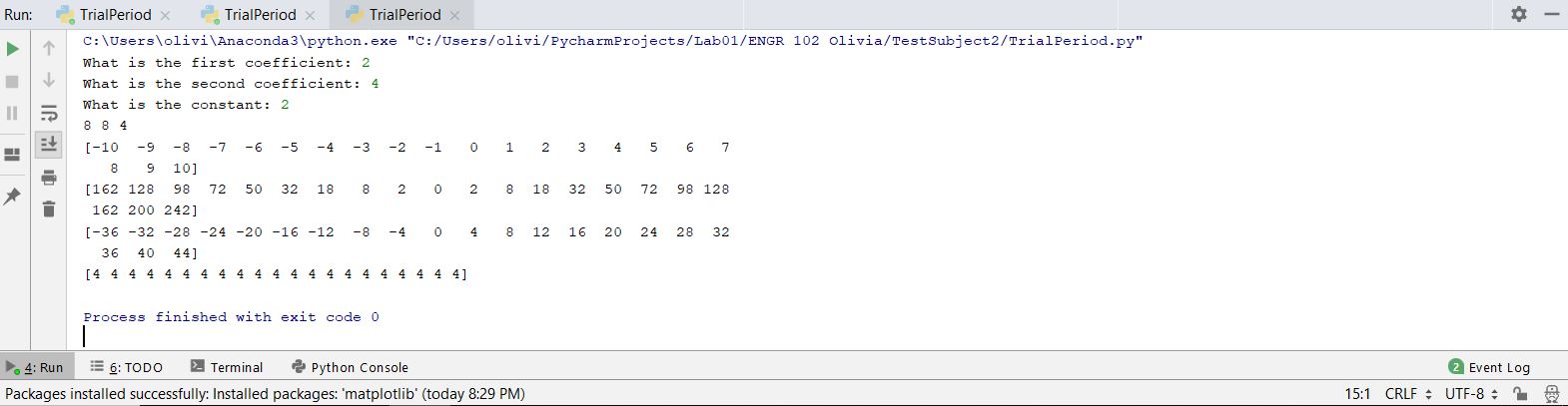
Follow the following process: Complete parts a and b before writing any code!

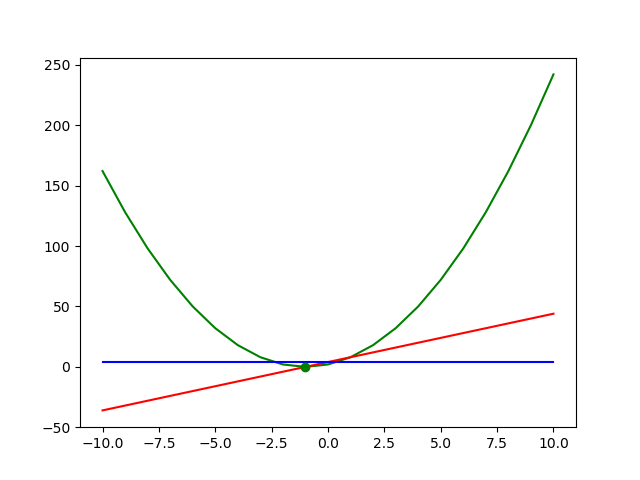
1. First, you will practice generating a bottom-up design:
   1. As a team, brainstorm what functionality you are likely to need in this program. For each basic task your team comes up with, where all the pieces needed are already known, write down a short function description, along with any input arguments needed, and what return value there will be, if any (i.e. construct a document, which will also be used for the subsequent design descriptions).
   2. Only do this for basic functions for which the steps you will need to follow are well-known or obvious. You can consider functions that will call other functions, as long as you have first designed the other function.
   3. Your goal in this part is not to generate a complete code design, but rather to identify functions to implement that are likely to be helpful in your program.
2. Next, create a top-down design of the program.
   1. Put together a document (can be added to the one from part (a) ) outlining the top-down design that your team comes up with.
   2. You should make use of the functions created in the bottom-up “phase”, above, to whatever extent seems helpful.
3. Next, code up your solution.
   1. You should create separate functions for each of the “nodes” in the top-down hierarchy that you created.
   2. You should also create separate functions for the features you identified in the bottom-up phase (a), above.
   3. Note that you might want to split up the coding responsibilities for the different functions among different team members. If you did a good job with design, the individual functions should be able to be coded up independently of one another.
   4. As a team, you will need to decide what variables you will use in the main program
   5. Be sure to include docstrings for all the functions you create

Code:

*# By submitting this assignment, all team members agree to the following:  
# “Aggies do not lie, cheat, or steal, or tolerate those who do”  
# “I have not given or received any unauthorized aid on this assignment”  
#  
# Names: Alexia Perez  
# Bethany Gawalis  
# Sam Lyzzaik  
# Tyler Scataglia  
# Section: 508  
# Assignment: Lab 12a  
# Date: 15-11-2018***from** math **import** \*  
**import** numpy **as** np  
**import** matplotlib.pyplot **as** plt  
**import** matplotlib  
**from** sympy **import** \*  
  
 *# Alternative program to be able to graph everything:  
  
# In this case we will not be using a flexible function, instead, we will only ask for the coefficients  
# and we will automatically make it a qudratic equation:*c1 = int(input(**'What is the first coefficient: '**))  
c2 = int(input(**'What is the second coefficient: '**))  
c = int(input(**'What is the constant: '**))  
  
**def** f(x):  
 **return** c1\*x\*\*2+c2\*x+c  
  
**def** fprime(x):  
 **return** c1\*2\*x+c2  
  
**def** fdoubleprime(x):  
 **return** c1\*2  
  
  
print(f(1),fprime(1),fdoubleprime(1))  
  
xval = np.array([-10,-9,-8,-7,-6,-5,-4,-3,-2,-1,0,1,2,3,4,5,6,7,8,9,10])  
yval = []  
yprimeval = []  
ydoubleprime = []  
**for** i **in** range(-10,11,1):  
 yval.append(f(i))  
 yprimeval.append(fprime(i))  
 ydoubleprime.append(fdoubleprime(i))  
  
y1 = np.array(yval)  
y2 = np.array(yprimeval)  
y3 = np.array(ydoubleprime)  
  
print(xval)  
print(y1)  
print(y2)  
print(y3)  
  
  
plt.plot(xval,y1,**'g'**)  
plt.plot(xval,y2,**'r'**)  
plt.plot(xval,y3,**'b'**)  
plt.plot(-1,0,**'go'**)  
plt.show()

Output:





**[40 points] Activity #2: Integrating a function – to be done as a team**

This activity is meant to familiarize you with the process of integration by computing area under a curve. As you may have learned by now in calculus, to integrate a function over a range of values, we can compute the area under the curve, when plotted in a graph. To do this, we essentially divide up the area under the curve into a number of simple shapes (like rectangles) and sum up the area of those shapes; as we use more and more of the shapes, we get a more and more accurate estimate of the area. For this example, we will use rectangles and trapezoids.

You are to write a program that computes the area under a function from one value to another. You should allow this to be an arbitrary function.

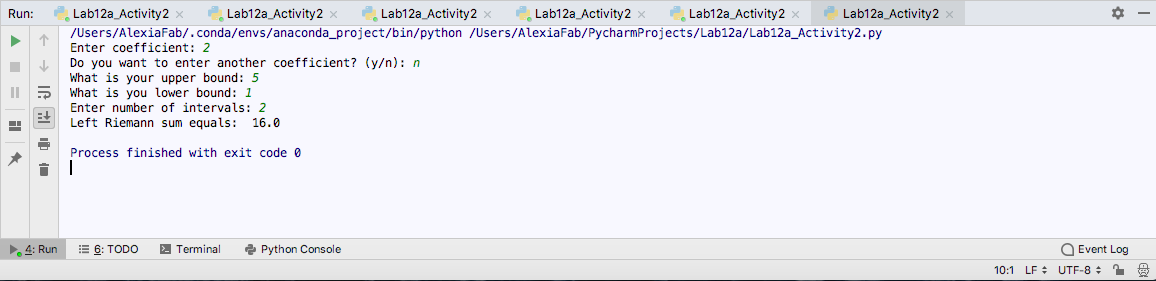
* Begin by outlining your program. Similar to part 1, you should identify any functions that you want to include (this program should be simpler than the one in activity #1). Put together a document listing the functions you will use in your program.
* You should take in as input a starting and ending value of the range that you want to integrate over.
  + You may, optionally, take in input for the function itself that you want to integrate. (If you choose to do this, see the note on “eval” in Activity #3.)
* You should integrate the function by dividing it into a number of elements, computing the area for each of those elements, and summing up those areas.
* Your program should begin by dividing the range into 10 elements, and then repeatedly increasing the number of elements until the area converges.
  + You may assume convergence if consecutive area calculations differ by less than 10-6.
* For each of the shapes, your program should report both the area computed, and how many iterations it took to achieve convergence.
* You should compute area using 4 different shapes/methods:
  + A rectangle whose height is the value of the function at the beginning of the interval
  + A rectangle whose height is the value of the function at the end of the interval
  + A rectangle whose height is the value of the function at the midpoint of the interval
  + A trapezoid with one parallel edge being the value of the function at the beginning of the interval, and another being the value of the function at the end of the interval.

For each of these, calculate the area for a particular subinterval –for the rectangles, the width will be the interval width, for the trapezpoid, that will be the “height” of the trapezoid.

Code:

*# By submitting this assignment, all team members agree to the following:  
# “Aggies do not lie, cheat, or steal, or tolerate those who do”  
# “I have not given or received any unauthorized aid on this assignment”  
#  
# Names: Alexia Perez  
# Bethany Gawalis  
# Sam Lyzzaik  
# Tyler Scataglia  
# Section: 508  
# Assignment: Lab 12a  
# Date: 19-11-2018***def get\_co**()**:** inp **= 'y'** co **=** []  
 **while** inp **== 'y':** A **=** float(input(**'Enter coefficient: '**))  
 co.append(A)  
 inp **=** input(**'Do you want to enter another coefficient? (y/n): '**)  
 **return** co  
  
  
co**=** get\_co()  
  
  
**def f**(*x*)**:** degree **=** len(co)  
 function **=** 0  
 **for** element **in** co**:  
 for** i **in** range(degree)**:** function **+=** element**\****x***\*\***(degree**-**i)  
 **return** function  
  
  
**def left\_Riemann\_sum**()**:** points **=** []  
 **for** i **in** range(int(Upper\_Bound))**:** pt **=** Lower\_Bound **+** i  
 points.append(pt)  
 Xi\_values **=** []  
 **for** i **in** range(0,int(len(points)**-**2),2)**:** Xi **=** points[i]  
 Xi\_values.append(Xi)  
 f\_xi\_list **=** []  
 **for** i **in** range(len(Xi\_values))**:** val **=** Xi\_values[i]  
 f\_xi **=** f(val)  
 f\_xi\_list.append(f\_xi)  
 s **=** 0  
 **for** element **in** f\_xi\_list**:** s **+=** int(element)  
 sum\_f\_xi**=** s  
 sum **=** delta\_n**\***sum\_f\_xi  
 **return** sum  
  
*# Output:*Upper\_Bound **=** float(input(**'What is your upper bound: '**))  
Lower\_Bound **=** float(input(**'What is you lower bound: '**))  
n **=** int(input(**'Enter number of intervals: '**))  
delta\_n **=** ((Upper\_Bound**-**Lower\_Bound)**/**n)  
print(**'Left Riemann sum equals: '**,left\_Riemann\_sum())

Output:

****

**[up to 55 points] Bonus/Challenge Activity #3: Expanding your code (challenge project – may be done over next week as a team for bonus credit)**

It is common when developing code that you think of additional things that a given program could do, but were not part of the original design. Generally, the better designed your original program, the easier it is to modify and expand it later in the process. Expanding a program in this way is often an example of bottom-up design: taking two or more things you already know how to do, and combining them to create something with more functionality.

For the program developed in activity #1, you can improve on it in various ways, possibly by incorporating code you have developed in previous weeks. As a team, you should take the program, and try to expand it in either or both of two ways. You should create a document outlining your plans before coding, and answering questions following the coding

Specifically, you should consider as a team how you can expand your code to do one or more of the following:

1. Rather than working only with polynomials, allow your program to work with an arbitrary function.
2. Instead of finding maxima and minima of the main function and the derivative curve by just searching through a list of points, you should find roots of the derivative curve (or of the second derivative curve), which will represent local minima and maxima.
3. Plot your curves over an arbitrary range of x and y values (you can ask the user, or determine these automatically)

Note that you may want to build on code developed in previous labs (especially lab 11) to assist you. In particular, you may want to make use of functions that compute a derivative numerically, and find a root (either by bisection or Newton’s method).

You should build any improvements incrementally; partial bonus will be awarded for progress toward the goals. In particular, note the following:

1. In prior work to find numerical derivatives, you would have a function that could evaluate a particular mathematical function at a value. You can do that here, but you will not have an analytical representation for the derivative function, so taking a second derivative may be more problematic. Here are two options:
   1. Represent your derivative function as a list of points. You can use the interpolation routines you have developed previously to evaluate that derivative function at any value. This would allow you to compute the derivative of the derivative (i.e. the second derivative) in the same way you evaluated the derivative itself.
   2. You can compute a second derivative directly from the original function (it is basically a sequence of differences – you will need to look up the process yourselves), and set up the second derivative curve that way.
2. Prior work has focused on finding only a single root in a range. For this expansion to work well, you would want to identify all roots in the range of the plot. You should feel free to begin your work by having a program that identifies only one root in the range. Once that is working, you may wish to try to write a function that can compute multiple roots in a range.
3. If you do both parts 1 and 2, then finding roots of a function which is known only by numberical differences will require a new approach. You can still find approximate roots of such a function, but will have to use interpolated values from the points.
4. You may want to consider the use of the python function “eval” (you will need to look up the details of how it works) to evaluate an arbitrary function that the user enters. You would ask the user to write the function as a python expression, get that input as a string, and then use eval to evaluate the expression.

Regardless of what you do, follow this process:

1. Use an incremental (“Pyramid style”) development approach. Add one thing at a time, and make sure it is working before adding something else.
2. You should be sure to break up your code into functions; do not write long sequences code.
3. For EACH feature added, put together a document with the following information:
   1. Prior to writing code, identify what existing functions you will be making use of both in the “main” program and from prior programs, what new functions, if any, need to be written, and what prior functions need to be modified.
   2. Following the code implementation, reflect: How, if at all, would you have designed prior parts of your program differently to allow this expansion? If it was very easy to add a new part of code in, your answer might be that your prior design worked well and was easy to expand in this way.
4. You should submit your code and your document for EACH feature added on. In the end, you may have multiple documents and multiple code samples for this project.