Numerical Analysis

Final task

Submission date: 15/2/2023 23:59

This task is individual. No collaboration is allowed. Plagiarism will be checked and will not be tolerated.

The programming language for this task is Python 3.7. You can use standard libraries coming with Anaconda distribution. In particular limited use of numpy and pytorch is allowed and highly encouraged.

Comments within the Python templates of the assignment code are an integral part of the assignment instructions.

You should not use those parts of the libraries that implement numerical methods taught in this course. This includes, for example, finding roots and intersections of functions, interpolation, integration, matrix decomposition, eigenvectors, solving linear systems, etc.

The use of the following methods in the submitted code must be clearly announced in the beginning of the explanation of each assignment where it is used and will result in reduction of points:

numpy.linalg.solve (15% of the assignment score)

(not studied in class) numpy.linalg.cholesky, torch.cholesky, linalg.qr, torch.qr (1% of the assignment score)

numpy.*.polyfit, numpy.*.*fit (40% of the assignment score)

numpy.*.interpolate, torch.*.interpolate (60% of the assignment score)

numpy.*.roots (30% of the assignment 2 score and 15% of the assignment 3 score)

All numeric differentiation functions are allowed (including gradients, and the gradient descent algorithm).

Additional functions and penalties may be allowed according to the task forum.

You must not use reflection (self-modifying code).

Attached are mockups of for 4 assignments where you need to add your code implementing the relevant functions. You can add classes and auxiliary methods as needed. Unittests found within the assignment files must pass before submission. You can add any number of additional unittests to ensure correctness of your implementation.

In addition, attached are two supplementary python modules. You can use them but you cannot change them.

Upon the completion of the final task, you should submit the four assignment files and this document with answers to the theoretical questions archived together in a file named <your ID>.zip

All assignments will be graded according to **accuracy** of the numerical solutions and **running time**.

Expect that the assignment will be tested on various combinations of the arguments including function, ranges, target errors, and target time. We advise to use the functions listed below as test cases and benchmarks. At least half of the test functions will be polynomials. Functions 3,8,10,11 will account for at most 4% of the test cases. All test functions are continuous in the given range. If no range is given the function is continuous in $[-\infty, +\infty]$.

1. $f_1(x) = 5$
2. $f_2(x) = x^2 - 3x + 5$
3. $f_3(x) = \sin(x^2)$
4. $f_4(x) = e^{-2x^2}$
5. $f_5(x) = \arctan(x)$
6. $f_6(x) = \frac{\sin(x)}{x}$
7. $f_7(x) = \frac{1}{\ln(x)}$
8. $f_8(x) = e^{e^x}$
9. $f_9(x) = \ln(\ln(x))$
10. $f_{10}(x) = \sin(\ln(x))$
11. $f_{11}(x) = 2^{\frac{1}{x^2}} * \sin\left(\frac{1}{x}\right)$
12. For Assignment 4 see sampleFunction.*

Assignment 1 (14pt):

Check comments in Assignment1.py.

Implement the function **Assignment1.interpolate(..)**.

The function will receive a function f, a range, and a number of points to use.

The function will return another "interpolated" function g. During testing, g will be called with various floats x to test for the interpolation errors.

Grading policy (10pt):

Running time complexity > O(n^2): 0-20%

Running time complexity = O(n^2): 20-80%

Running time complexity = O(n): 50-100%

The grade within the above ranges is a function of the average absolute error of the interpolation function at random test points. Correctly implemented linear splines will give you 50% of the assignment value.

Solutions will be tested with $n \in \{1,10,20,50,100\}$ on variety of functions at least half of which are polynomials of various degrees with coefficients ranging in $[-1,1]$.

Question 1.1: Explain the key points in your implementation (4pt).

In this code, I used the cubic Bezier interpolation. First, I generated a set of n points (and n-1 sub-intervals) to interpolate over the interval [a, b]. To find the vectors A and B, I used the formulas we learned in class and Thomas algorithm. I created for each sub-interval cubic Bezier curve equation, according to each sub-interval's edge points and the appropriate a and b (from the vectors A and B). In addition, I created the function g(x). The function receives x and try to find the sub-interval in which the x is located (by looking for the first edge point the x is smaller from). Then, in order to find the value of y, I derivate the appropriate sub-interval's cubic Bezier curve equation, and found the value of t using the appropriate x's values (which we know) and the Secant method(and then was able to find y's value). Lastly, I return g.
I choose Cubic Bezier interpolation over other interpolation methods because it provides a smooth curve that can be used to approximate a wide range of functions. It is especially useful for interpolating functions that have curvature, as it provides a more accurate representation of the function than linear or polynomial interpolation. In addition, with this method I was able to get a running time complexity of O(n).

Assignment 2 (14pt):

Check comments in Assignment2.py.

Implement the function **Assignment2.intersections(..)**.

The function will receive 2 functions- f_1, f_2, and a float maxerr.

The function will return an iterable of approximate intersection Xs, such that:

$$\forall x \in X, |f_1(x) - f_2(x)| < maxerr$$

Grading policy (10pt): The grade will be affected by the number of correct/incorrect intersection points found and the running time of **Assignment2.intersections(..)**.

Question 2.1: Explain the key points in your implementation (4pt).

In this code, I used the Regula Falsi method. First, I created a helper function g(x), which return the subtraction of f1(x) and f2(x). I calculated 'val' as the width of the interval [a, b] divided by 100. Then, t is initialized as a + val and x1 and x2 are set to a and t respectively. I check for each sub-interval if its edge point(a and t) are greater than 0. If it's true, then a and t are updated to a + val and t + val, respectively, and the process continues. Else, then it means that there is a possible intersection point within the sub-interval [a, t], use Regula Falsi on this sub-interval (the error I checked is equal to maxerr), and put the value I get in the list X. Finally, I returned the list X.
I divided the interval [a,b] into smaller interval to get more precise intersection points while I use the Regula Falsi method . Also, finding the relevant intervals by checking the sign of the edge points will prevent unnecessary calculations. In addition, I decide to use Regula Falsi method, because its convergence rate tend to be faster than Bisection method's convergence rate , and it is more robust than the Secant method.

Assignment 3 (36pt):

Check comments in Assignment3.py.

Implement a function **Assignment3.integrate(…)** and **Assignment3.areabetween(..)** and answer two theoretical questions.

Assignment3.integrate(…) receives a function f, a range, and several points to use.

It must return approximation to the integral of the function f in the given range.

You may call f at most n times.

Grading policy (10pt): The grade is affected by the integration error only, provided reasonable running time e.g., no more than 5 minutes for n=100.

Question 3.1: Explain the key points in your implementation of Assignment3.integrate(…). (4pt)

First, I create the helper function called simpson(a, b, f, n), which calculates the definite integral of a function(f) using the Simpson's Rule on n-1 intervals(n points).
In the function "integrate", I checked 3 cases: if n=1 or (n-1)=1, I used midpoint rule to calculate the integral; if n-1 is even, I used the helper function "simpson" on n-1 intervals(n points); if n-1 is odd, I used the helper function "simpson" on n-2 intervals(n-1 points). Finally, I returned the appropriate integral.
For this assignment, I receive the restriction to call f at most n times. If n=1 or (n-1)=1, I can't use the Simpson or Trapezoidal rule, because this rules requires to call to f at least two times(and in case (n-1)=1, according to my implement I use the Simpson's Rule on n-2 intervals, meaning in this case on 0 intervals and it is not possible), so in this case I used the midpoint rule, which allows me to call to f just one time(use just the midpoint of the interval [a, b]). In addition, I checked if n-1 is odd or even, because I can use the Simpson's Rule on an even number of intervals. So, if n-1 is odd, I used the Simpson's Rule on n-2 intervals(and n-1 points), which is an even number of intervals.

Assignment3.areabetween(..) receives two functions f_1, f_2 .

It must return the area between f_1, f_2 .

In order to correctly solve this assignment you will have to find all intersection points between the two functions. You may ignore all intersection points outside the range $x \in [1,100]$.

Note: there is no such thing as negative “area”.

Grading policy (10pt): The assignment will be graded according to the integration error and running time.

Question 3.2: Explain the key points in your implementation of Assignment3. areabetween (…) (4pt).

First, I found all intersection points between the functions. If the is less than 2 points, this means there is no area between the functions, and I returned "nan". Else, for each pair of edge points I found

the integral of each function between the points and found the absolute value of the subtraction of the two areas. I summed all values and returned the result.

Question 3.3: Explain why is the function $2^{\frac{1}{x^2}} * \sin\left(\frac{1}{x}\right)$ is difficult for numeric integration with equally spaced points? (4pt)

The function $2^{\frac{1}{x^2}} * \sin\left(\frac{1}{x}\right)$ is difficult for numeric integration with equally spaced points because it has a singularity at x = 0, which means that the function approaches infinity as x approaches zero. This singularity makes it very challenging to use traditional numerical integration methods, such as the trapezoidal rule or Simpson's rule, which rely on evaluating the function at equally spaced points. When the integration interval includes the singularity, the function exhibits rapid oscillations near x = 0, making it even more difficult to accurately approximate the integral using equally spaced points. The high-frequency oscillations require a very small spacing between the evaluation points to capture accurately, which increases the number of points needed.
For example, consider we are trying to integrate this function over the range [0, 1], we would find that using only a few equally spaced points leads to a large error in the result. We would need to use many more points, or use a different integration method, to get an accurate result.

Question 3.4: What is the maximal integration error of the $2^{\frac{1}{x^2}} * \sin\left(\frac{1}{x}\right)$ in the range [0.1, 10]? Explain. (4pt)

To answer this question, let's look on the error of the Simpson's rule and the error of the trapezoidal rule. I will not check the error of the midpoint rule because in this case (finding the maximal error), it will always be lower than the trapezoidal rule.
Maximal error of the Simpson's rule: $E = -\frac{(b-a)^5}{180n^4} * f''''(\varepsilon)$.
Maximal error of the trapezoidal rule: $E = -\frac{(b-a)^3}{12n^2} * f''(\varepsilon)$.
The errors are proportional to the second and the fourth derivative of the function. In the case of the function $2^{\frac{1}{x^2}} * \sin\left(\frac{1}{x}\right)$, the derivatives are unbounded near x=0, which means that if we decide that $\varepsilon = 0.1$(the closest value in this range to 0), the errors become arbitrarily large.
In addition, in order to maximize the errors, we will choose the minimal n possible for each rule: n=1 for the trapezoidal rule and n=2 for the Simpson's rule.
I used online calculators to find the second and fourth derivative:
Second:

$$\frac{\left(2\cos\left(\frac{1}{x}\right)x^3 + (6\ln(2)-1)\sin\left(\frac{1}{x}\right)x^2 + 4\ln(2)\cos\left(\frac{1}{x}\right)x + 4\ln^2(2)\sin\left(\frac{1}{x}\right)\right)\cdot 2^{\frac{1}{x^2}}}{x^6}$$

Fourth:

$$\frac{\left(24\cos\left(\frac{1}{x}\right)x^7 + (120\ln(2)-36)\sin\left(\frac{1}{x}\right)x^6 + (216\ln(2)-12)\cos\left(\frac{1}{x}\right)x^5 + \left(300\ln^2(2)-84\ln(2)+1\right)\sin\left(\frac{1}{x}\right)x^4 + \left(192\ln^2(2)-8\ln(2)\right)\cos\left(\frac{1}{x}\right)x^3 + \left(144\ln^3(2)-24\ln^2(2)\right)\sin\left(\frac{1}{x}\right)x^2 + 32\ln^3(2)\cos\left(\frac{1}{x}\right)x + 16\ln^4(2)\sin\left(\frac{1}{x}\right)\right)\cdot 2^{\frac{1}{x^2}}}{x^{12}}$$

In addition, I checked and discovered that $\varepsilon = 0.1$ is indeed the value where I get values of the derivatives that maximized the errors:
$f''(0.1) = -1.644 * 10^{36}$
$f''''(0.1) = -4.031 * 10^{42}$
Now, let's calculate each error:

Simpson: $E = \frac{-(9.9)^5 * (-4.031 * 10)^{42}}{180 * 2^4} = \frac{3.834 * 10^{47}}{2880} = 1.33 * 10^{44}$

Trapezoidal: $E = \frac{-(9.9)^3 * (-1.644) * 10^{36}}{12 * 1^2} = \frac{1.595 * 10^{39}}{12} = 1.33 * 10^{38}$

We can see that the error of the Simpson's rule is bigger, therefore the maximal integration error of the $2^{\frac{1}{x^2}} * \sin\left(\frac{1}{x}\right)$ in the range [0.1, 10] is $1.33 * 10^{44}$.

Assignment 4 (14pt)

Check comments in Assignment4.py.

Implement the function **Assignment4.fit(…)**

The function will receive an input function that returns noisy results. The noise is normally distributed.

Assignment4A.fit should return a function g fitting the data sampled from the noisy function. Use least squares fitting such that g will exactly match the clean (not noisy) version of the given function.

To aid in the fitting process the arguments a and b signify the range of the sampling. The argument d is the expected degree of a polynomial that would match the clean (not noisy) version of the given function.

You have no constrains on the number of invocation of the noisy function but the maximal running time is limited. Additional parameter to **Assignment4.fit** is maxtime representing the maximum allowed runtime of the function, if the function will execute more than the given amount of time, the grade will be significantly reduced.

Grading policy (10pt): the grade is affected by the error between g (that you return) and the clean (not noisy) version of the given function, much like in Assignment1. 65% of the test cases for grading will be polynomials with degree up to 3, with the correct degree specified by d. 30% will be polynomials of degrees 4-12, with the correct degree specified by d. 5% will be non-polynomials

Question 4.1: Explain the key points in your implementation. (4pt)

The function "fit" starts by calculating the value of f at a and measuring the time taken. If the time taken is more than 0.97 * maxtime, the method returns a lambda function that returns linear function (y =x). Otherwise, the function calculates the number of data points that can be calculated within the maximum allowed time. If this number is less than the degree of the polynomial plus one, the function calculates the coefficients of the polynomial function we are looking for using the available points and the helper function "least_square". Else, I took (d+1) equally spaces xs, and for each x I calculated n-ys (n is the limit of times I can use f on each x in the time limit). Then, I calculates the average of the y-values of each x and use it (in addition to the xs) and "least_square" to calculate the coefficients.
Finally, the function returns a lambda function that takes in a value x and returns the value of the polynomial at x, using the coefficients obtained from "least_square".
By using the least square method in this assignment, I ensure that the fitted function closely follows the underlying trend in the data, while also minimizing the effects of any noise or outliers in the data. In addition, Taking the mean of multiple noisy measurements can help reduce the effect of random errors, which can arise due to various factors. By taking the mean, the effect of these random errors is reduced, and a more accurate estimate of the underlying function value is obtained, which can improve the accuracy of the fitted polynomial.

Assignment 5 (27pt).

Check comments in Assignment5.py.

Implement the function **Assignment5.area(…)**

The function will receive a shape contour and should return the approximate area of the shape. Contour can be sampled by calling with the desired number of points on the contour as an argument. The points are roughly equally spaced.

Naturally, the more points you request from the contour the more accurately you can compute the area. Your error will converge to zero for large n. You can assume that 10,000 points are sufficient to precisely compute the shape area. Your challenge is stopping earlier than according to the desired error in order to save running time.

Grading policy (9pt): the grade is affected by your running time.

Question 4B.1: Explain the key points in your implementation. (4pt)

As we can see, the function receives AbstractShape.contour, which return an array of consecutive points on the shape contour (the number of points depends on the argument contour receives). So, to calculate the area of the shape, I decided to use the shoelace formula. In this formula, we take vertices of a polygon (in our case take a set of points on the shape contour) and calculate its area using the formula:
$area = 0.5 * |\sum_{i=1}^{n-1} x_i y_{i+1} + x_n y_1 - \sum_{i=1}^{n-1} x_{i+1} y_i - x_1 y_n| \ (n \ is \ the \ number \ of \ points)$. I decided to use this formula because it's easy to implement and it is more efficient than other algorithms of area's calculation.
First, I take 10 points from the contour (by creating a value n=10), and created the value 'area' which initialized to be 0. Then, in a while loop, I calculated the area of the shape using the 10 points and shoelace formula, save the area as 'sec_area', and calculated the absolute value of the difference between 'area' and 'sec_area'.
The while loop continues to iterate until the absolute value of the difference were smaller or equal to maxerr, and each iteration I calculated the area with ten times bigger number of points(n) than the previous iteration, and calculated the absolute value of the difference between the current area and the area of the previous iteration. In addition, I stop the while loop if n became bigger 10000.

Implement the function **Assignment4.fit_shape(…)** and the class **MyShape**

The function will receive a generator (a function that when called), will return a point (tuple) (x,y), a that is close to the shape contour.

Assume the sampling method might be noisy- meaning there might be errors in the sampling.

The function will return an object which extends **AbstractShape**
When calling the function **AbstractShape.contour(n)**, the return value should be array of n equally spaced points (tuples of x,y).

Additional parameter to **Assignment4.fit_shape** is maxtime representing the maximum allowed runtime of the function, if the function will execute more than the given amount of time, the grade will be significantly reduced.

In this assignment only, you may use any numeric optimization libraries and tools. Reflection is not allowed.

Grading policy (10pt): the grade is affected by the error of the area function of the shape returned by Assignment4.fit_shape.

Question 4B.2: Explain the key points in your implementation. (4pt)

The function starts by receiving a point from the sample function and uses that point to calculate the interval time taken to receive a single point. If the interval time is greater than 0.97*maxtime (the time limit), the function returns a MyShape object with an area of 0 (because we will receive just one point and it's not enough for the area's calculations).
Then, the function stores the points I can get in the time limit in a list. The center of all the points (and the shape) is calculated by taking the average of all the points. The function uses the KMeans algorithm from the scikit-learn library. KMeans is an unsupervised machine learning algorithm used for clustering. The main objective of KMeans is to group similar data points together into clusters. It works by defining a number of clusters (also known as k) and then finding the centroid of each cluster. The function cluster the points into 36 different clusters. Then, I use the helper function 'sort_points', which sorts the centers of the clusters based on their polar angle relative to the center point of all the points.
Finally, the function calculates the area of the shape using the sorted centers of the clusters. The area is calculated using the shoelace formula, which involves adding and subtracting the product of the x and y coordinates of the sorted centers. The result is a MyShape object with the calculated area (I implement MyShape in a way that it receives an area of the shape, save it as the field of the class, and when I use the method 'area', it returns the area).
I decided to use KMeans algorithm because it helps to identify the main features of the shape by grouping the noisy points together that are close to each other. It helps to reduce the noise in the data and provide a more accurate representation of the shape contour and provides a good approximation of the data distribution. In addition, I sorted the center points based on their polar angle relative to the center point of all the points because in order to use the shoelace formula properly, the points need to be sorted in counterclockwise order, which can be achieved by using this sort.