Numerical Analysis

Final task

Submission date: 22/2/22 23:59 (strict).

This task is individual. No collaboration is allowed. Plagiarism 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.

You should not use those parts of the libraries that implement numerical methods taught in this course (unless explicitly stated otherwise in the instructions of the particular assignment)**.** This restriction 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 deduction of points. Failure to announce the use of any restricted functions will result in disqualification of the assignment.

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

(not studied in class) numpy.linalg.cholesky, torch.cholesky, linalg.qr, torch.qr (10% 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)

numeric differentiation functions are allowed!

numpy.linalg.inv, scipy.linalg.inv, torch.inverse, and all other external libraries for matrix inversion (20% of the assignment score)

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

You must not use reflection (self-modifying or self-inspecting 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. BUT! existing unit tests are provided for demonstration and to encourage you to write additional tests as you go. You can add any number of additional unittests to ensure correctness of your implementation. Passing only the existing unittests does not ensure that your code will not fail in all cases. It is your responsibility to test your code and ensure that it is stable. You should add 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 five assignment files and this document with answers to the theoretical questions. The archive should not contain folders, but only the submission files!

Assignments will be graded according to **error** of the numerical solutions and **running time**. Some assignments have required specific error bounds – they will be graded according to running time. Some assignments limit the running time – they will be graded according to error. For all executions there is 2 minutes running time cap after which the execution will be halted.

Every assignment will be AUTOMATICALLY tested on a number of different functions and different parameters. It may be executed multiple times on the same function with the same parameters. Every execution will start with a clean memory. Any exception throwed during an execution will render the execution invalid and nullify its contribution to the grade. **Test your code!!!**

Any disqualification of an assignment (e.g. due to unannounced use of restricted functions) or an execution (e.g. due to exception) will not contribute to the grade regardless the effort put in the development.

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 – add additional unittests with implementations of these functions. At least half of the test functions will be polynomials. Functions 3,8,10,11 will account for at most 5% 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):

(10pt) Implement the function **Assignment1.interpolate(..)** following the pydoc instructions.

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:

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

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

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

Running time complexity will be measured empirically as a function of n.

The grade within the above ranges is a function of the average relative 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 $n \in \{1,10,20,50,100,200,500,1000\}$ on variety of functions at least half of which are polynomials of various degrees with coefficients ranging in $[-1,1]$.

Restricted functions I used:

None

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

At first I implemented my answer with Lagrange. But because of it complexity to greater inputs I changed to Bezier's Interpolation algorithm learned In class, with the following steps:

- Splitting the range given to parts with linespace and sampling the points.
- In order to getting Bezier curves for each 2 set of points I used a helping function which build a linear system following the 4 rules to create coefficients matrix (which also Triagonal matrix), And In order to solve it efficiently I used Thomas algorithm.
- After finding a and b from the equations, for each 2 points I create the Bezier curve by the formula of B(t)= $(1-t^3)P0 + 3(1-t)^2 t * P1 + 3(1-t) * t^2 * P2 + t^3 P3$
  After then for a given point as input, I search with a loop the corresponding function, place the t which I compute approximately with its' x value comparing to the 2 sampled points.

Assignment 2 (14pt):

(10pt) Implement the function **Assignment2.intersections(..)** following the pydoc instructions.

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:

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

$$\forall x_i x_j \in X, |x_i - x_j| > maxerr$$

Grading policy: The grade will be affected by the number of correct and incorrect intersection points found, the running time of **itr = Assignment2.intersections(..)** followed by **list(itr).**

Restricted functions I used:

none

(4pt) Question 2.1: Explain the key points in your implementation in particular explain how did you address the problem of finding multiple roots.

Ass2 was fun, I created a new function of the substitution of the given functios, an small number for safety(called eps) , and a flag for 3 different states.
The helping function for computing the root (with the new function the the given error) was regula Falsi.
And for the iteration, I implemented using the next principles.
I set the first point (a) as the left border, and the right as left+maxerr+eps. On each iteration I check if the left border is legal following the maxerr rules, and the last value of the answers' array (and add to the array if so) . If its not, I use a regula falsi with the borders, and If the returned root is legal, I add it to the array. In any case, I set the left border as the point I added to the array, or else – the right border. And on each iteration I increase the right border by maxxerr+eps (in order so do less checks on the left border condition).

- Improvements : I used flags in order to improve some checks and to find more points, and their concept is to check if point were found in some iteration, and It didn't find in the following one – it goes back and try to find again with a smaller error (maxerr/2).

Assignment 3 (31pt):

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

(5pt) Assignment3.integrate(…) receives a function f, a range, and a number of points n.

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: The grade is affected by the integration error only, provided reasonable running time e.g., no more than 2 minutes for n=100.

Restricted functions I used:

none

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

- In this implementation I used the Simpson's 1/3 rules (with some small changes for optimizations), and in order to maintain the question restrictions, I checked how many times f is used on various tests, and decided to sub n by 1 if it's even, and by 2 if its odd.
- After that I calculate the integration using Simpson's algorithm, and in a special case (of b<a) I subtracted n once more, because of the algorithm behavior.

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

Restricted functions I used:

None

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

In order to implement that part, I used ass2 intersections functions In order to get array of points with a small maxerr (0.002). (smaller then that my program will crawl due the implantation of 2). After that I iterated over the intersection points, calculating the absolute value of the subtraction between the area of every two points In the array for each given function(using 3.1) , and summing the into my answer variable, which I returned after finishing the iterations.

(4pt) 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?

In the function provided above around the area in the left border of the domain (where x is close to 0), the oscillation's space over small interval returns extreme values for given x's (f(x) is getting close to infinity), And the intervals are the same, so it causes getting repeating huge values. To deal with such functions and improve the results we should break our intervals into smaller pieces to decrease the error.

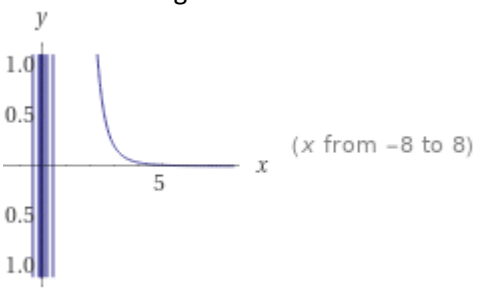
As summary, The given function $2^{\frac{1}{x^2}} * \sin\left(\frac{1}{x}\right)$ is difficult for numeric integration because it is not smooth' over an interval, so giving it equally spaced points isn't good enough. To get accurate and good results, the space points should be chosen wisely in the implementation.

(4pt) 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.

According to Simpson's rule error to find the maximum error we need to find K in the given range, according to the maximum of value in the range for the 4'th derivative.

Simpson's rule error bound $|E_S| \leq \frac{K(b-a)^5}{180n^4}$ $\left|f^{(4)}(x)\right| \leq K$

After observing the behavior of the function 4'th derivative in "wolframalpha":



After taking 0.1 as global extreme point I got a really huge error. It seems that the function doesn't have any upper bound in the given range and even strive for infinity, so as conclusion the maximum error is high and strive for infinity as well - which support the explanation of 3.3 for the difficulty of getting accurate results.

Assignment 4 (14pt)

(10pt) Implement the function **Assignment4.fit(…)** following the pydoc instructions.

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

Assignment4.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 invocations of the noisy function but the maximal running time is limited. Invocation of f may take some time but will never take longer than 0.5 sec.

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 execution will not contribute to the grade causing significant deduction. You should consider the risk of failure vs gains in accuracy when you get close to the time limit.

Grading policy: 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. 60% 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. 10% will be non-polynomials with random $d \in [1..12]$.

Restricted functions I used:

None

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

Here are the following points:

- I created an array for points and added the two edges of the range.
- Then for reasonable amount of time, (0.66 of the max time) I kept expanding the points array by adding between each two points another point in the middle with a helping function, as long as I'm not passing the time limit.
- Then, as learned in class, I created the A array of the x values and arranged them in a matrix by their degrees (accidently created the transpose first, but it was helpful because I need both).
- Created the b vector of the f(x) values.
- Then in order to solve the equation then we learned in practical sessions of AT*Ax=AT*B, I created both of the equations' side using multiplication, and to solve the matrix I used another helping function of inversing the matrix, and applied it to (AT*A), and multiplied it with ATB, giving me the vector of the solutions – which I transformed to a polynomial.

Assignment 5 (27pt).

(9pt) Implement the function **Assignment5.area(…)** following the pydoc instructions.

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 that according to the desired error in order to save running time.

Grading policy: the grade is affected by your running time.

Restricted functions I used:

none

(4pt) Question 5.1: Explain the key points in your implementation.

I set a starting time for safety, and a step of 200, and iteration's counter.
On each iteration, I set length to be the number of points I sample using step*2, (meaning I sample first 400, then 800,1600, etc), and I save the area using the helping function to calculate it -
For this question I used a helping function the calculating the area with trapezoidal method. If the changes are smaller than max error, I return the last area calculated, otherwise – if I passed the time limit of 0.5, I stop and return the area of 10,000.

(10pt) Implement the function **Assignment5.fit_shape(…)** and the class **MyShape** following the pydoc instructions.

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 should 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). When calling the function **AbstractShape.area()**, the return value should be the area of the shape. You may use your solution to **Assignment5.area** to implement the area function.

Additional parameter to **Assignment5.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 execution will be halted.

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

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

There are no restricted functions. The use of any library is allowed.

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

For this question I did not make it implementing the full solution. I implemented the Abstract Shape to get a set of points and area in the constructor, so can return contour, sample and area as asked. The area given was computed after following steps. I sample 10,000 points I computed a mid x and y according to the sum of the sample points, and used it to sort them by clockwise order, by angle and distance. After that I used the 5.1 trapezoidal function to compute its' area.