Due Friday, November 20, 24:00

Problem 13 (Bonus)

Wikipedia says the centered finite difference stencil for the m^{th} derivative with convergence $\mathcal{O}(h^n)$ has $2p+1 = 2\left\lfloor \frac{m+1}{2} \right\rfloor - 1 + n$ coefficients a_p which can be calculated by the solution of the linear system

$$\begin{pmatrix} 1 & 1 & ... & 1 & 1 \\ -p & -p+1 & ... & p-1 & p \\ (-p)^2 & (-p+1)^2 & ... & (p-1)^2 & p^2 \\ ... & ... & ... & ... & ... \\ ... & ... & ... & ... & ... \\ ... & ... & ... & ... & ... \\ (-p)^{2p} & (-p+1)^{2p} & ... & (p-1)^{2p} & p^{2p} \end{pmatrix} \begin{pmatrix} a_{-p} \\ a_{-p+1} \\ a_{-p+2} \\ ... \\ ... \\ ... \\ a_p \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ ... \\ m! \\ ... \\ 0 \end{pmatrix},$$

where $\lfloor x \rfloor$ is the *floor* function which is the biggest integer smaller than or equal to x. Similarly, the coefficients a_i for the d^{th} derivative calculated by an arbitrary stencil $(s_1\ s_2\ \dots s_N)$[1] with $d < N$ are given by

$$\begin{pmatrix} s_1^0 & \cdots & s_N^0 \\ \vdots & \ddots & \vdots \\ s_1^{N-1} & \cdots & s_N^{N-1} \end{pmatrix} \begin{pmatrix} a_1 \\ \vdots \\ a_N \end{pmatrix} = d! \begin{pmatrix} \delta_{0,d} \\ \vdots \\ \delta_{i,d} \\ \vdots \\ \delta_{N-1,d} \end{pmatrix}$$

Prove both of these statements.

There is a Python package, `findiff`, that provides stencils for any derivatives with any accuracy in any dimensions. We will not use it, but it could be handy in your future research. This package is not part of anaconda, so installing it requires some care. Do not play with it for now unless you are willing to go through some shell commands.

Problem 14

In this problem we will consider functions f represented by their values at regularly spaced discrete points x_i. This means the function will be represented by a vector $\vec{f}$ whose components are $f(x_i)$. We will use the function $1 + \sin(\pi x + \frac{\pi}{4})$ on the interval $[-1\ 1]$.

(a) Implement `numerical_derivative(fx, a, b)` that receives, in order, a vector representation of a function, the initial sampling point and the final sampling point. It returns a vector with the numerical derivative of the function at the same sampling points. Use a $\mathcal{O}(h^2)$ accurate stencil, and make sure you keep the same accuracy at the end points by adjusting the stencils appropriately.

Let the length of `fx` be $n+1$, and apply the above derivative operator with $n = 2^{15}$ (i.e. $h = 2^{-14}$). You can easily calculate the analytical result, so plot it together with the numerical one, and also make a separate plot that shows the difference. Let's call this numerical derivative f'_h, and remember that $f'_h = f' + e_h$ where f' is the true derivative and e_h is the truncation error.

In general you will not know the actual function in its analytical form, and will only have access to the sample points. A good way to *estimate* the error in the derivative is looking at the difference of numerical derivatives with different n values, e.g. $f'_h - f'_{h/2}$ gives you an idea about your error. How is this difference related to e_h and $e_{h/2}$?

[1] In this notation the centered stencils would be $(-ph\ -(p-1)h\ \dots 0 \dots (p-1)h\ ph)$

Take the numerical derivative for $n = 2^{16}$ and look at its difference from $n = 2^{15}$. And then repeat this by doubling n (halving h) and looking at the difference from the previous case until the roundoff error dominates instead of the truncation error. How do you understand roundoff error starts to dominate? Calculate $f'_h - f'_{h/2}$ for each case. Plot all the differences in a graph, remember that these differences are a good measure of your error. It will be hard to see all the curves directly, but if you scale each successive $f'_h - f'_{h/2}$ by a factor of 4, all curves should agree aside from the one with roundoff error domination. Explain this observation.

It is enough to return the final plot and the explanation, you need not show the intermediate steps we ask you to follow in this paragraph.

(b) Implement `numerical_integral(fun, a, b, method='simpson')` with similar arguments as before which returns a vector with the numerical integral of the function from `a` to the corresponding sampling point, i.e. the i^{th} element of the output vector is the integral from a to $a + \frac{(b-a)}{n}i$ (index i starting from 0). Have two options for the integration method, *simpson* and *trapezoid*.

Repeat the plots and error analyses in part (a) for each method. When did you start to see the roundoff error? Compare your observations to those in part (a), and explain any differences. What is the convergence you observe? Does it agree with the theory? What scaling factor did you need for the difference between successive h values so that they agreed?

Problem 15 (Bonus)

Let's continue working on the previous problem using the same function in the same interval.

(a) Implement the quadratically convergent second derivative as `numerical_2derivative(fx, a, b)`, make sure to take care of the endpoints. Repeat the tests.

The second derivative is nothing but the first derivative of the first derivative at the analytical level. Apply the first derivative twice to calculate the second derivative in an alternative way. Does this method converge *everywhere* as expected?

(b) Differentiation is supposed to be the inverse operation of integration. First take the integral of the function with the trapezoid rule, and then take the derivative, and finally look at the difference from the function you began with. Plot this difference for successively halved values of h until you start to see the effects of the roundoff error. Repeat, but this time with Simpson's rule. Comment on your observations.

(c) Knowing the convergence properties of our truncation error enabled us to calculate it. It actually also enables us to improve on it! Using what you know about the dependence of e_h on h, obtain an $\mathcal{O}(h^4)$ numerical derivative result using two calls to `numerical_derivative(fx, a, b)` (remember that this function calculates the derivative with $\mathcal{O}(h^2)$ error). Can you get the improved convergence everywhere, if not why?

Problem 16

Let's assume you need to integrate and differentiate the $d_L(z)$ data from Problem 11 for some reason. You only know the function from its values at discrete points, so numerical calculus is a must. You might quickly realize that there is a problem: We cannot control where we observe the supernovae, so $d_L(z)$ is sampled at irregularly spaced points whereas the formulae we are familiar with are based on equal intervals.

(a) Find the three-point stencil formulas for the first derivative for irregularly spaced data, i.e. find α_k in

$$f'(x_i) \approx \alpha_{i-1} f(x_{i-1}) + \alpha_i f(x_i) + \alpha_{i+1} f(x_{i+1})$$

Hint: This is straightforward if you solved Problem 13.

Implement this in Python as `numerical_derivative_irregular(fx,x)` which takes the derivative of a function whose values that are sampled at `x` are `fx`, and returns an array of the same size that contains the above numerical approximation to the derivative of the function. You can simply use two-point stencils at the endpoints,[2] and use the average d_L for repeated z values with different d_L. Plot your result together with the derivative of a cubic spline fit to $d_L(z)$ data, and comment on the result.

(b) Wikipedia has the formula for the composite Simpson's rule for irregularly spaced data

$$\begin{aligned}\int_a^b f(x)\,\mathrm{d}x &= \sum_{i=0}^{N/2-1} (\alpha_i f_{2i+2} + \beta_i f_{2i+1} + \eta_i f_{2i}) \\ \alpha_i &= \frac{2h_{2i+1}^3 - h_{2i}^3 + 3h_{2i}h_{2i+1}^2}{6h_{2i+1}(h_{2i+1}+h_{2i})} \\ \beta_i &= \frac{h_{2i+1}^3 + h_{2i}^3 + 3h_{2i+1}h_{2i}(h_{2i+1}+h_{2i})}{6h_{2i+1}h_{2i}} \\ \eta_i &= \frac{2h_{2i}^3 - h_{2i+1}^3 + 3h_{2i+1}h_{2i}^2}{6h_{2i}(h_{2i+1}+h_{2i})} \ .\end{aligned}$$

I suggest you derive this formula for your numerical analysis skills, but you don't have to. Implement the Python function `numerical_integral_irregular(fx,x)` which takes the definite integral of a function whose values that are sampled at `x` are `fx`. The boundaries of the integral are the least and greatest elements of `x` (you can assume `x` has ascending order). If `x` has an even number of elements, you can use the trapezoid rule in the last interval[2]. Plot the integral $\int_0^z dz\ d_L(z)$ as a function of z together with the integral of a cubic spline fit to $d_L(z)$ data, and comment on the result.

As you may expect, SciPy already has this function implemented as `scipy.integrate.simps`. Compare your functions performance to this in terms of accuracy and speed. Remember to look for builtin functions for the task at hand even if we ask for a custom implementation.

Problem 17

Finite difference is the essence of numerical differentiation and integration in many respects, and we will use it for differential equations. However, if the only thing we are after is finding the definite integral of an analytically known function, there are methods that are more effective than Newton-Cotes formulas in certain cases. Quadrature is the general name for calculating an integral by looking at the values of the integrands at certain points (hence the name of the builtin function in SciPy), so Simpson's method is technically an example of a quadrature rule where we sample the function a large number of times. There are also methods that rely on fewer points and some underlying mathematical structure such as Gaussian quadrature. The approximate integration formulas are still similar

$$\int_{-1}^{1} dx f(x) \approx \sum_{i=1}^{n} w_i f(x_i) \ ,$$

where w_i are called the *weights*. In the most basic Gauss-Legendre quadrature x_i are the roots of the n^{th} Legendre polynomial

$$P_n(x) = \frac{1}{2^n n!}\frac{d^n}{dx^n}(x^2-1)^n \ ,$$

and $w_i = \frac{2}{(1-x_i^2)[P_n'(x_i)]^2}$. Why go through the trouble of finding these roots? Because then you can get an accurate approximate result with a few dozen points rather than the gazillion points you need for Simpson's

[2] This will reduce your accuracy, but we are not after high precision in this problem.

rule. This was especially valuable before the advent of digital computers, you can tabulate the roots of some fixed polynomial once, and then use these to do a few dozen arithmetic operations compared to thousands or millions you may need for a naive Simpson's method implementation (though there are tricks with Simpson's method as well). You do not need to know the details of this magic, but it is related to the fact that P_n form an orthogonal basis for polynomials in the interval $[-1\ 1]$

$$n \neq m \implies \int_{-1}^{1} dx P_n(x) P_m(x) = 0$$

If you read the documentation for `scipy.integrate.quad`, you will see that it uses Gaussian quadrature by default rather than some Newton-Cotes formula.

(a) Implement your `quad_custom(fun,a,b)` which uses quadrature to take integrals, i.e. it works similarly to `scipy.integrate.quad(fun,a,b)` we used in problem set 4. Life is short to write your own Legendre functions let alone find their roots, so use the functions in the Python module `numpy.polynomial.legendre`. The basic object in this module is not a simple Legendre polynomial, but rather a sum of a bunch of them with coefficients. For example, you find the roots as follows:

```
>>> import numpy.polynomial.legendre as leg
>>> leg.legroots((1, 2, 3, 4)) # 4L_3 + 3L_2 + 2L_1 + 1L_0, all real roots. L_i are Legendre polynomials
array([-0.85099543, -0.11407192,  0.51506735])
```

So, to find the roots of a single Legendre polynomial, the argument of `leg.legroots` should be a vector with many zeros and a single one.

Test your `quad_custom` on the function from Problem 14 for n values ranging from 10 to 20, and find the error by comparing to the analytically known result. How many sampling points in Problem 14 gives the error we have here for $n = 20$? [3]

(b) Note that Newton-Cotes methods or Gauss-Legendre quadrature are not capable of taking the integral $\int_0^\infty e^{-x} dx$.[4] How can you take this, or any other integral which has to be performed on an infinite interval? Explain your reasoning, and take this integral using any of the methods we implemented in this problem set. *Hint: My usual hint.*

[3] This is probably a good place to introduce you to Rosetta Code which is a source that explains and lists sample implementations of many basic mathematical methods in various programming languages. Feel free to check it out, but my personal experience with it is scarce, so I cannot endorse it without reservations. Do not take their code as the golden standard (even though it may be). Reading actual code is one of the best ways to learn programming structures in a language. Note that Rosetta Code implements its own Legendre polynomials and root finders for Gauss-Legendre quadrature, but you can rely on NumPy packages for these as instructed above.

[4] Strictly speaking, Gauss-Legendre quadrature is only defined on the fixed interval $[-1\ 1]$, but by the time you figure out the issue for infinity, you would have already seen how to address this as well.