Ph 20.2 – Introduction to Numerical Techniques: Numerical Integration

Due: Week 3 -v20170314-

This Assignment

The purpose of the last assignment was to familarize you with the basic procedure of writing numerical code with Python and `numpy` and making plots using `matplotlib`. In this assignment our tools will be the same, but the emphasis will be on making sure we're using them correctly—that is, debugging and sanity-checking code. The subject of this week's assignment, numerical integration, is interesting in its own right, but it's also useful for this purpose because we can prove mathematically how it should behave. That means if you observe that your code *isn't* behaving that way there must be an error in the code (or the proof, in principle—but in this case that shouldn't be an issue!). In future projects you won't always be able to prove the expected behavior in as much detail as we have here, but there will often still be ways to validate your code and convince yourself that the results make sense. Think of this as the coding equivalent of dimensional analysis.

Completing this assignment requires doing some math, writing some code, and making plots and a writeup. The math portion and the coding portion can be done mostly independently of each other, at least once you've remembered, looked up, or derived the extended Simpson's formula; the writeup will combine the results of both (and give you lots of practice writing equations in LaTeX). You may want to focus on the coding part in lab so you can ask your TA questions. When implementing (11) and its Simpson's equivalent remember that you can use `numpy` to do operations on an entire array at once. You'll want to generate the set of points x_i using an appopriate `numpy` function, then evaluate the function at these points, then sum these values with the appropriate weightings depending on which method you're using. For this last step it's useful to know how to operate on particular subsets of an array; the keyword to look up is "(extended) array slicing." All of these steps are straightforward in principle but in practice there are many subtle ways to get them wrong; that's why this assignment is about debugging!

Numerical Integration

In the last assignment you made your acquaintance with what we could call the *standard process* of scientific programming (in its Python version): write code and debug it, by adding functionality incrementally, and by testing simpler units before you put them together; run your code to produce data files; plot those files; try to figure out what is going on. In this assignment you will be introduced to the first (and perhaps the simplest) of many numerical techniques: *numerical integration*.

You may ask why you should learn about it when there are many web-accessible `Python` modules to do the job, or alternatively a software system such as *Mathematica* (which we will explore later in the Ph20/21/22 sequence) can easily step in with `Integrate` or `NIntegrate` (for integrals that cannot be done symbolically). The simple answer is that you should indeed learn to use tools like *Mathematica*; however, many of the computational problems that you will encounter in your career will require you to write your own code modules. The primary purpose of this course is to give you an introduction to both approaches: using standard tools when they are available, and writing your own programs when necessary. The deeper answer is that even to use

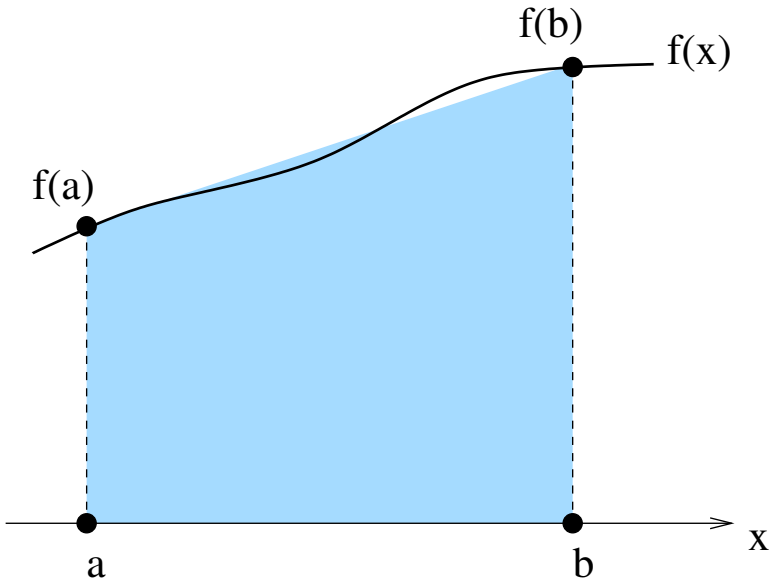

Figure 1: Trapezoidal rule approximation of an integral

the standard tools, you must have an understanding of the principles on which they are based, so that you can anticipate their limits and trace their performance.

There are many resources that provide information on numerical algorithms. A landmark book on the subject is *Numerical Recipes*. It provides a reasonably comprehensive and readable introduction to many of the most important numerical algorithms. You should definitely familiarize yourself with the content of *Numerical Recipes* and if you continue on in your computational work, the book may very likely end up in your library (either digital or hardcopy).

But first, some math!

Your assigned readings for this week provide background about numerical integration. Here I am just giving you the bare necessities for this week's assignment. But you *are* expected to go deeper, with the help and advice of your TA.

We are concerned with finding an approximate expression for the definite integral

$$I = \int_a^b f(x)\,dx. \tag{1}$$

Since this integral is the area under the function $f(x)$ in the interval $[a, b]$, we might think of approximating I as

$$I \simeq H\left(\frac{f(a)}{2} + \frac{f(b)}{2}\right) \equiv I_{\text{trap}}, \tag{2}$$

where $H = b - a$. This is known as the *trapezoidal rule*, because it approximates I by the area of the trapezoid with vertices $\{a, 0\}$, $\{a, f(a)\}$, $\{b, f(b)\}$, $\{b, 0\}$, replacing (in effect) the smooth function $f(x)$ with the segment running from $\{a, f(a)\}$ to $\{b, f(b)\}$ (see Figure 1).

How good is this approximation? We can find out by writing $f(x)$ as a Taylor sum centered in a (see your favorite calculus textbook, or `http://mathworld.wolfram.com/TaylorSeries.html`),

$$f(x) \;=\; f(a) + f'(a)(x-a) + \frac{f''(a)}{2!}(x-a)^2 + \frac{f'''(a)}{3!}(x-a)^3 + \cdots$$

$$= \quad f(a) + f'(a)(x-a) + \frac{f''(\eta)}{2!}(x-a)^2 \qquad \text{with } x, \eta \in [a,b], \tag{3}$$

where the exclamation mark denotes the factorial, and the last term in the second line is the *Lagrange remainder*. By elementary definite integration, we get

$$I = f(a)H + f'(a)\frac{H^2}{2!} + f''(\eta)\frac{H^3}{3!} \quad \text{with } \eta \in [a,b]. \tag{4}$$

We can also use Eq. (3) to approximate $f(b)$, and thus rewrite $I_{\rm trap}$ as

$$\begin{aligned} I_{\rm trap} = f(a)\frac{H}{2} + f(b)\frac{H}{2} &= f(a)\frac{H}{2} + \left(f(a) + f'(a)H + \frac{f''(\eta)}{2!}H^2\right)\frac{H}{2} \\ &= f(a)H + f'(a)\frac{H^2}{2} + f''(\eta)\frac{H^3}{2\cdot 2!} \qquad \text{with } \eta \in [a,b], \end{aligned} \tag{5}$$

which differs from Eq. (4) by

$$I_{\rm trap} - I = f''(\eta)\frac{H^3}{4} - f''(\eta)\frac{H^3}{6} = f''(\eta)\frac{H^3}{12}, \tag{6}$$

Thus, we say that the trapezoidal rule is *locally* of third order in H, since

$$I = I_1 + O(H^3) \tag{7}$$

[we shall see in a moment what we mean by locally]. Do higher order approximations exist? Sure indeed: by using the value of $f(x)$ at the third point $c = (a+b)/2$, we can construct *Simpson's rule* (nothing to do with Matt Groening, I'm afraid),

$$I \simeq I_{\rm simp} \equiv H\left(\frac{f(a)}{6} + \frac{4f(c)}{6} + \frac{f(b)}{6}\right); \tag{8}$$

the coefficients are chosen to match the terms of the analogs of Eqs. (4) and (5), written with Taylor terms up to $\propto f''''(x)H^5$. In fact, by virtue of some cancellations, Simpson's rule turns out to be of *fifth* (rather than fourth) order in H,

$$I = I_{\rm simp} + O(H^5). \tag{9}$$

Obviously the error in the estimation of integrals improve as we go to higher and higher order in H, but we quickly run into severe algebraic-manipulation pain when we try to compute the fractional coefficients that sit in front of the function values $f(a)$, *etc.* There is a simpler way to get smaller errors: reduce the value of H. The way to do this is to divide the interval $[a,b]$ into N subintervals, and use the approximation formulas for each interval. We define $h_N = (b-a)/N$, and $x_0 = a$, $x_1 = a + h_N$, $x_2 = a + 2h_N$, $\ldots$, $x_N = b$. We then find, using the trapezoidal rule, that

$$\begin{aligned} \int_a^b f(x)\,dx &= \int_{x_0}^{x_1} f(x)\,dx + \int_{x_1}^{x_2} f(x)\,dx + \cdots + \int_{x_{N-1}}^{x_N} f(x)\,dx \\ &\simeq h_N\left(\frac{f(x_0)}{2} + \frac{f(x_1)}{2}\right) + h_N\left(\frac{f(x_1)}{2} + \frac{f(x_2)}{2}\right) + \cdots + h_N\left(\frac{f(x_{N-1})}{2} + \frac{f(x_N)}{2}\right) \qquad (10) \\ &\equiv h_N\left(\frac{f(x_0)}{2} + f(x_1) + f(x_2) + \cdots \frac{f(x_N)}{2}\right). \qquad (11) \end{aligned}$$

The last line represents the *extended trapezoidal formula*: it might not seem very different from Eq. (10), but from the computational standpoint it is much more effective, because it avoids the double evaluation of the function $f(x)$ at each point $x_1, \ldots, x_{N-1}$. The *global error* of Eq. (11) is approximately

$$-f''(\xi)\frac{h_N^3}{12} \cdot N = -(b-a)f''(\xi)\frac{h_N^2}{12}, \quad \text{with} \quad \xi \in [a,b], \tag{12}$$

so the extended trapezoidal rule is globally of second order in h_N, and increasing the number of subintervals by 2 improves the accuracy of the result by 4. (This property of the extended trapezoidal rule makes this rule important in Romberg integration and other methods). It should now be clear what we meant by local error vs. global error.

The Assignment

1. Derive an extended formula for Simpson's formula [Eq. 4.1.13 of *Numerical Recipes*; you can find physical copies of *NR* in the lab or digital versions of it online]. Also, show that its local error is $O(H^5)$, and that its global error is $O(h_N^4)$. If you just skimmed over the preceding math, note that we've derived all three of these things for the trapezoidal rule above; you can use these derivations as an outline for your own Simpson's rule derivations with minor modifications (e.g. taking the Taylor expansion to higher order).

2. Write a general python function to integrate an externally defined function using the extended trapezoidal rule. The function should take the arguments *func*, a, b, and N. Here *func* is the name of a function to integrate, a and b are the extrema of integration, and N is the number of subintervals to use.

   As you may have seen in this week's readings, Python's syntax is very generous in allowing you to pass a function name as an argument.

   Use numpy arrays instead of for-loops whenever possible.

3. Write a python routine to integrate an externally defined function using Simpson's extended formula. Have the subroutine take the same arguments as in Part 2.

4. Using your routines, evaluate the integral of e^x between 0 and 1. If you know how to use *Mathematica*, get the expected result to many digits of accuracy from *Mathematica* (use `Integrate` and `N`). If not, you may use the value: 1.7182818284590452354. Or, better, recognize that this is $e - 1$ and just use `np.e - 1`! Next compare the accuracy of your methods for several values of N, the number of subintervals. Construct a plot that shows the convergence rate of the error for each method, as you increase N.

5. Sanity check: does your convergence plot show the global error behavior you expect from the discussion/derivation above? If you can't tell from your plot, make a more illustrative one. If it doesn't show the behavior you expect, there's at least one error in your code! Once you're satisfied your code is correct, test its limits by going out to large values of N, say $N \sim 5000$ (you might find `np.logspace` useful here). You should find that at some point the error stops decreasing. What do you think is going wrong? (More discussion on this next week.)

6. Write a general purpose routine that integrates an externally defined function to a specified level of relative accuracy: the routine should evaluate Simpson's formula for $N = N_0$, $2N_0$,

$4N_0, \ldots$ (with a reasonable N_0), until the relative difference between successive approximations,

$$\left| \frac{I_{\rm simp}(N = 2^k N_0) - I_{\rm simp}(N = 2^{k+1} N_0)}{I_{\rm simp}(N = 2^k N_0)} \right|, \tag{13}$$

is less than the accuracy requested. Try this routine on e^x, and compare with previous results. Next, try it on a function of your choice.

7. Take a look at `http://docs.scipy.org/doc/scipy/reference/tutorial/integrate.html`. Use `scipy.integrate.quad` and `scipy.integrate.romberg` and compare the results with parts 3-5 of this assignment.

8. $\star$ (optional - for those with further interest in numerical algorithms). Experiment with writing your own version of *Romberg's method*, discussed in *Numerical Recipes*. This is a rather powerful approach that can be generalized to other numerical techniques. The basic idea is the following: several trial estimates of the integral are made using the extended trapezoidal rule, breaking up the integration interval into more and more subintervals for each successive trial. The subroutine keeps track of the sequence of estimates and then predicts what the estimate would be in the limit of an infinite number of subintervals (or alternatively zero step size, $h = 0$). It turns out that Romberg's method is an extremely fast and accurate method for numerical integration in a large variety of circumstances.