Overview

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 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

\[ I = \int_a^b f(x) \, dx. \]  

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

\[ I \approx H \left( \frac{f(a)}{2} + \frac{f(b)}{2} \right) \equiv I_{\text{trap}}, \]

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).
Figure 1: Trapezoidal rule approximation of an integral

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),

\[
f(x) = f(a) + f'(a)(x-a) + \frac{f''(a)}{2!}(x-a)^2 + \frac{f'''(a)}{3!}(x-a)^3 + \ldots
\]

with \( x, \eta \in [a,b] \),

(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

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

(4)

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

\[
I_{\text{trap}} = f(a)\frac{H}{2} + f(b)\frac{H}{2} = f(a)\frac{H}{2} + \left( f(a) + f'(a)H + \frac{f''(\eta)H^2}{2!} \right)\frac{H}{2}
\]

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

(5)

which differs from Eq. (4) by

\[
I_{\text{trap}} - I = f''(\eta)\frac{H^3}{4} - f''(\eta)\frac{H^3}{6} = f''(\eta)\frac{H^3}{12},
\]

(6)

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

\[
I = I_1 + O(H^3)
\]

(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),

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

(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 $H$,

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

(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

$$\int_a^b f(x) \, dx = \int_{x_0}^{x_1} f(x) \, dx + \int_{x_1}^{x_2} f(x) \, dx + \cdots + \int_{x_{N-1}}^{x_N} f(x) \, dx$$

$$\simeq h_N \left( \frac{f(x_0)}{2} + \frac{f(x_1)}{2} \right)+h_N \left( \frac{f(x_1)}{2} + \frac{f(x_2)}{2} \right)+\cdots+h_N \left( \frac{f(x_{N-1})}{2} + \frac{f(x_N)}{2} \right);$$

(10)

$$\equiv h_N \left( \frac{f(x_0)}{2} + f(x_1) + f(x_2) + \cdots + f(x_N) \right).$$

(11)

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

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

(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]. Also, show that its local error is $O(H^5)$, and that its global error is $O(h_N^4)$.

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. Next compare the accuracy of your methods for several values of $N$, the number of subintervals. Construct a plot that shows the difference convergence rate of the error within each method, as you increase $N$.

5. 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,

$$\left| \frac{I_{\text{simp}}(N = 2^kN_0) - I_{\text{simp}}(N = 2^{k+1}N_0)}{I_{\text{simp}}(N = 2^kN_0)} \right|,$$

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.

6. 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.

7. ★ (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.