PHAS0030: Further Practical Mathematics & Computing
Session 3: Integration & Differentiation

David Bowler

January 18, 2019

In this session, we will consider discretization in computational physics, in different forms. First, we will consider types of grid that can be used, and the circumstances in which they are appropriate. Then we will look at approximating differentiation using functions evaluated at discrete points. We will finish with the inverse operation: integrating a function on a grid.

Contents

1 Objectives

The objectives of this session are to:

- discuss different types of grids that are used
- understand the finite difference method
- explore ways to perform numerical integration

2 Review of Session 2

In the second session, we discussed:

- Accuracy and precision in computational physics
- Building models of physical problems
- Root finding algorithms
- Approaches to optimisation of functions

By now, you should be confident with the creation, manipulation and use of Numpy arrays, and feel that you understand how to use different kinds of flow control to implement algorithms. You should be happy writing functions (and becoming familiar with the idea of passing a function as a parameter to another function). We will not introduce any more significant elements of the Python language from here on, though it will be important for you to continue to practise these things. We will gradually expand our knowledge of library routines in Numpy and Scipy.

3 Grids

Discretisation is (almost) inevitable on a computer: we generally have to break space and/or time into small steps. We may have the analytic form of some functions, but we will not in general be able to solve the equations that we write down analytically. Using a discrete representation, we will be able to solve (almost) all problems that there are in physics, at some level of approximation[1] In space, if there is a restricted range for a variable which takes finite steps, we can think of this as forming a grid: $x = x_0 + i \times \Delta x, i = 0 \rightarrow N - 1$. The grid does not need to be linear, and if using more than one dimension, the axes do not need to be orthogonal. In time, we will most often move forward in time in discrete steps as an approximation to an integral when solving differential equations. In all cases, the *grid spacing* is key to both accuracy and computational effort[2].

[1] This is a very grand claim, and there are many areas where the approximation is quite drastic. There are some classes of problem which are known to be effectively impossible to solve on a *classical* computer.

[2] And note that the effort will depend on the dimensionality of the problem: the computer effort will scale as N^D for N grid points in each dimension, D.

3.1 *Types of grid*

The simplest grid is uniformly spaced, with orthogonal axes. If the spacing is different in different directions, then we call the resulting grid orthorhombic. This is the simplest grid to use, and is very common in computational physics. It is easily mapped onto a Numpy array. If we have a 3D grid with $(l \times m \times n)$ grid points then the corresponding array would be `a[l,m,n]`. If you want to define corresponding Cartesian coordinates then they must have the same dimensionality as the array, though they will contain redundant information[3]

[3] In 2D, `x[l,m]` will change only with `l`, and will have the same value for all values of `m`.

However, we will find that different grids are useful in different circumstances (just as different coordinate systems are used). It is easy to define polar coordinate grids both in two and three dimensions. Typically these are most useful in situations which match the coordinate symmetry. For instance, atomic calculations almost always use spherical polar coordinates: the angular parts of solutions can be written in terms of spherical harmonics, leaving only a radial grid to be defined[4]

[4] The angular terms can be calculated analytically; this is rather rare.

In an atomic calculation, the radial grid is often not linear, but logarithmic, often defined as:

$$r_i = \beta e^{\alpha(i-1)} \tag{1}$$

This requires a little care when considering differentials: if you are solving an equation that contains $\partial y/\partial r$ you will need to account for $dr/di = \alpha r$ in the computational implementation.

Non-orthogonal grids are quite common: a triangular (or hexagonal) mesh is very helpful for certain situations, and is easily defined. In this case, the indices of the Numpy array will give the multiple of the basis vectors rather than mapping directly onto space (see further work for an example). It is possible to work with *unstructured* grids, though neither these nor non-orthogonal grids are likely to feature in the problems in this course.

3.2 *Exercises*

In class

1. Use `np.logspace` to create an array x with twenty elements that runs from 10^{-3} to 1. Create an array y of twenty zeros (or ones if you prefer) and plot y vs x.
2. Now make a new plot of the same arrays using `plt.semilogx`. I found `plt.grid` helpful here (on both axes).
3. The Archimedes spiral is defined as $r = a + b\theta$. Make a polar plot (`plt.polar(theta,r)`) for $0 \leq \theta \leq 6\pi$. Experiment with values of a and b.

4. The logarithmic spiral is rather different: $r = aexp(b\theta)$, where a and b are *different* variables to those in the Archimedes spiral. Make a polar plot for this.

Further work

1. With the array that you created Q1 (in class), make plots of $1/x$ on linear axes (`plt.plot`), semi-log axes (`plt.semilogx`) and log-log axes (`plt.loglog`). Think carefully about the advantages and disadvantages of each of these approaches.

2. Make Cartesian plots for the Archimedes and logarithmic spirals, calculating x and y explicitly from r and θ. Experiment with using points rather than lines for the plot: would a logarithmic grid or some other grid make more sense for the logarithmic spiral with points?

3. (*More challenging; do not worry if you cannot complete this*) We can create a non-orthogonal grid by defining vectors $\mathbf{a} = (1, 0)$ and $\mathbf{b} = (0.5, \sqrt{3}/2)$ and a mesh whose points are found at $i\mathbf{a} + j\mathbf{b}$ for i and j integers. Create one array `pos` (dimensions [2,N,N] where N is the grid size) to store the x and y components of the grid positions and calculate them (it's easiest to loop over i and j and use the expression above). Create a 2D function such as $\cos(x) \times \sin(y)$ (remember that you can use `pos[0]` to access the entire 2D array of x values) and plot it using `plt.contourf(x,y,surface)`. You can visualise the grid simply with `plt.plot(pos[0],plt.plot[1]`.

4 *Differentiation*

We start by recalling the formal mathematical definition of a differential

$$\frac{df}{dx} = \lim_{\delta x \to 0} \frac{f(x + \delta x) - f(x)}{\delta x} \tag{2}$$

Equivalently, if we consider a Taylor expansion for a function about a point x, then we see:

$$f(x + h) = f(x) + hf'(x) + \ldots \tag{3}$$

This suggests that the differential of a function can be approximated rather easily to first order. Computationally, we can implement this by choosing a *finite* difference, Δx, in place of δx, and checking for the convergence as $\Delta x \to 0$.

The definition given above is for the forward difference, but there is no reason for this direction to be special. We can consider the backward difference:

$$\left.\frac{df}{dx}\right|_x \simeq \frac{f(x) - f(x - \Delta x)}{\Delta x} \tag{4}$$

These definitions both show a limited accuracy; if we make return to the Taylor expansion of $f(x)$, we can see why:

$$f(x+\delta x) = f(x) + \delta x f'(x) + \frac{\delta x^2}{2!} f''(x) + \frac{\delta x^3}{3!} f'''(x) + \dots \quad (5)$$

Re-arranging, it is clear that the error in the derivative is first order in δx; there is a linear dependence, related to the curvature:

$$f'(x) = \frac{f(x+\delta x) - f(x)}{\delta x} - \frac{\delta x}{2!} f''(x) - \dots \quad (6)$$

Is this best approximation that can be made? No - it is possible to make the error second order in the difference ($\propto \Delta x^2$) rather easily, by considering the points $x + \Delta x$ and $x - \Delta x$:

$$\left.\frac{df}{dx}\right|_x \simeq \frac{f(x+\Delta x) - f(x-\Delta x)}{2\Delta x} \quad (7)$$

This approach, known as finite differences, can be extended to higher orders of accuracy, though the higher the order, the larger the spread of points required which can increase the computational effort. The accuracy of the approximation will generally improve as the value of Δx decreases, though care is needed: you will find in the exercises that if it is too small, then rounding error starts to affect the results.

Finite differences are normally used on grids, where the grid spacing is very unlikely to be small enough to reach round-off error. If you have a function stored on an array of grid points (i.e. an array storing the values of f) then the differential can be found rather quickly using the function `np.roll`, which shifts an array to the right or the left. If you do this, you must be rather careful about the boundaries: do you have a periodic function, or do you have to use a different approach at the boundaries?

You can also think about the calculation of a finite difference in terms of a matrix operating on a vector: the vector contains the elements of the array for the function, and the matrix will be based around the diagonal. The matrix for a forward difference can be written:

$$\frac{1}{\Delta x}\begin{pmatrix} -1 & 1 & 0 & 0 & 0 & \dots \\ 0 & -1 & 1 & 0 & 0 & \dots \\ 0 & 0 & -1 & 1 & 0 & \dots \\ 0 & 0 & 0 & -1 & 1 & \dots \\ \vdots & \vdots & \vdots & & \ddots & \end{pmatrix} \quad (8)$$

We can also go beyond the first differential by combining formulae from previous levels of differentiation. The second-order centred difference formula is given by:

$$f''(x) \simeq \frac{f(x+\Delta x) - 2f(x) + f(x-h)}{h^2} \quad (9)$$

(You can find this by considering centred first order differences for $x + \Delta x/2$ and $x - \Delta x/2$.)

Implementation of finite differences is rather simple, but there is a library function: `numpy.gradient`. The first argument is an array of the function that you wnat to differentiate; you can pass just this (in which case the spacing is assumed to be 1), or a spacing as a scalar, or an array the x-coordinate. If your function is more than one dimensional, then a single value for the spacing is applied to all directions; you can also specify an array of spacings, or an array of coordinates of appropriate size.

4.1 Exercises

In class

1. Create a function that implements Eq. (7), taking the function $f(x)$, x and Δx as inputs and returns the approximation to the differential
2. Test it on the sine function, using `np.cos` to check. Plot the difference between the approximation and the analytic differential for different values of Δx. Try using `np.logspace` to set up an array of values for Δx, and see if `plt.loglog` is useful. [Note: if you take it below about 10^{-5} then you will see interesting behaviour]
3. Now implement a function for the *centred* difference and do the same analysis. Does this behave as expected?

Further work

1. Create an array of x values from $0 \rightarrow 2\pi$ with spacing $\Delta x = 2\pi/500$. Create an array holding $\cos(x)$ using `np.cos`. Now calculate the finite difference of the array using `np.roll` and compare to the exact result found using `np.sin`. [The shift equivalent to $+\Delta x$ in `np.roll` is an index of -1. Make sure that you understand why. Also think about why we chose the value of Δx.]
2. Write a function to calculate the second derivative using the centred formula. Write another function that combines two first derivative FD functions and compare the results (you may wish to experiment with two forward differences vs forward and back).

5 Integration

Just as we can approximate a differential with the value of a function at different points, so we can approximate an integral by summing over the values of the function at different points with appropriate weights for the points.

At the simplest level, we replace the area under the curve with a series of rectangles, with the height taken from the left hand side of the rectangle:

$$\int_a^b f(x)dx \simeq \sum_{i=0}^{N-1} f(a+i\Delta x)\Delta x \tag{10}$$

where we must have $b - a = N\Delta x$. This echoes a formal definition of integration (where the width of the rectangle is taken to zero)[5]. We can improve on the simplistic approach here but replacing rectangles with trapeziums (trapezium rule):

[5] Note that in this case only, the value of f(b) is not used; in all other cases it will be. This comes from the simplicity of this approximation.

$$\int_a^b f(x)dx \simeq \Delta x \left(\frac{f(a)}{2} + \sum_{i=1}^{N-1} f(a+i\Delta x) + \frac{f(b)}{2} \right) \tag{11}$$

(Note that this is actually the same as the rectangle rule if you replace the height of the left hand side of each rectangle with the average height.)

A still better approximation comes by fitting quadratic curves through three neighbouring points (this is known as Simpson's rule). The resulting formula requires an *odd* number of points (if we run from 0 to N then N must be even) and applies *different* weights to different points:

$$\int_a^b f(x)dx \simeq \frac{\Delta x}{3} \left(f(a) + \sum_{i=1}^{N-1,\text{odd}} 4f(a+i\Delta x) + \sum_{i=2}^{N-2,\text{even}} 2f(a+i\Delta x) + f(b) \right) \tag{12}$$

Of course, there are also more sophisticated approaches; Gaussian quadrature is one well-known version, which for n points returns an estimate that is exact for polynomials up to degree $2n - 1$, but does not have evenly spaced points.

In multiple dimensions it is often fastest to use the equivalent of the rectangle rule (simply summing over values of the function, and scaling by the area or volume of the grid point), though the Monte-Carlo approach (which we will cover in Session 8) is also very effective. It is also worth considering the symmetry of the situation: if you have spherical symmetry then the angular terms can be integrated analytically, leaving only a one-dimensional radial numerical integral.

5.1 *Library routines*

As we have seen before, there are often efficient implementations of standard numerical techniques in libraries. Scipy provides routines for numerical integration in the integrate module (you would use `from scipy import integrate` to import). There are simple routines (`integrate.trapz` and `integrate.simps`) as well as more complex approaches.

The functions `trapz` and `simps` both take the same arguments:

- `y` An array of function values (effectively $f(x_i)$ for all values of i)

- `x` An *optional* array of the values of x_i
- `dx` An *optional* value of step size (defaults to 1.0)

They both return the value of the integral. Remember that Simpson's rule requires an odd number of points (from $0 \rightarrow N$ with N even).

The general 1D integration function is `quad(f,a,b,args=()`[6] which takes arguments:

- `f` is the function
- `a`, `b` are the limits
- `args` is an *optional* tuple

If your function is called simply as `f(x)` then ignore the `args` parameter. If your function requires some parameters, so that it is called as `f(x,c,d)` then you can pass values as `args=(c,d)`.

`quad` returns two numbers: the value of the integral, and an upper bound on the error in the integral.

[6] This is an interface to a low-level FORTRAN library, QUADPACK, that provides general-purpose quadrature—the formal name for numerical integration.

5.2 *Exercises*

In class

1. Create a simple integration function that uses the rectangle formula, Eq. (10). It will need to take a function, two limits and a number of points (or spacing) as arguments.
2. Calculate the definite integral of a simple function (choose something that you can solve analytically in the first instance, like $2x$ which will integrate to x^2) and investigate the effect of the spacing (width).
3. Now calculate the integral of $x\cos(x)$ from 0 to 1.
4. Implement the trapezium rule and do the same calculation. You should be able to find the exact answer using integration by parts.

Further work

1. Implement Simpson's rule, and integrate $x\cos(x)$ from 0 to 1. By looking at the formulae, consider which is the best approximation in terms of accuracy vs effort.
2. Use the library routine `quad` from `scipy.integrate` and see how accurate it is. Try adding an argument to your $x\cos(x)$ function, and passing it to `quad`.