PHAS0030: Computational Physics
Session 3: Integration & Differentiation

David Bowler

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.

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

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.

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: $x = x_0 + i \times \Delta x, i = 0 \rightarrow N - 1$ (though note that you should *not* us a loop to do this, but something like `np.arange`). 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].

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 created with a command like `a = np.array((l,m,n))`. As we have already seen, `np.meshgrid` allows us to create 2D (or 3D) arrays containing appropriate Cartesian coordinates.

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[3]

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

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

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

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

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.

Exercises

In class

1. Use `np.logspace` to create an array x with twenty elements that run from 10^{-3} to 1. Create an array y of twenty zeros (or ones if you prefer) and plot y vs x with points (the point here is to see how the grid works). Use `fig = plt.figure` and `ax = fig.add_subplot(1,2,1)` to create a 1×2 array of subplots (you will fill in the other plot in the next question). NB You will need to use `%matplotlib notebook` to be able to update a figure between different cells.

2. Now add a plot to your figure (remember that the final argument to `ax2 = fig.add_subplot` should now be 2, for the second subplot) of the same arrays using `ax2.semilogx`. I found `ax2.grid` helpful here (on both axes) to visualise what is happening. How do the two plots compare in terms of clarity?

3. The Archimedes spiral is defined in polar coordinates as $r = a + b\theta$. Create a new figure using `fig = plt.figure(figsize=(9,3))` (though you may change the figure size to suit your preference) and add a polar subplot using `ax = fig.add_subplot(1,2,1,polar=True)`. Make a polar plot for $0 \leq \theta \leq 6\pi$, calculating r from the definition, using `ax.plot(theta,r)` (note that the arguments are **not** the standard way around!)[4]. Experiment with values of a and b.

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

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).

[4] Note that you can produce a polar plot in a simple way using `plt.polar(theta,r)`.

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],’bo’)`.

Differentiation

We start by recalling the formal mathematical definition of a differential

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

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

$$f(x+\delta x) = f(x) + \delta x f'(x) + \cdots$$

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 $\Delta x \to 0$. Note that the differential is actually defined at a point $x + \Delta x/2$. We define:

$$\left.\frac{df}{dx}\right|_{x+\Delta x/2} = \frac{f(x+\Delta x) - f(x)}{\Delta x}.$$

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:

$$\left.\frac{df}{dx}\right|_{x-\Delta x/2} \simeq \frac{f(x) - f(x-\Delta x)}{\Delta x}$$

If you implement them (which you will, below), you will find that these definitions both show a limited accuracy; if we make return to the Taylor expansion of $f(x)$, we can see why:

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

Re-arranging for $f'(x)$, we find that the error in the derivative is *first order* in δx; there is a linear dependence, related to the curvature:

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

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 $x - \Delta x$:

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

The differential is now evaluated at x. Now consider an array representing the values of a function at a set of evenly spaced grid points (say $i\Delta x$). We can evaluate the derivative of the function at the grid points using this approach (in this case we would replace $x + \Delta x$ and x with grid points $(i + 1)$ and i).

This overall 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? This question of boundaries is very important throughout computational physics (and in a more general way, boundary conditions are often the most important quantities to consider when solving equations).

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. We would have $\mathbf{f}' = \underline{\underline{D}}\,\mathbf{f}$. The matrix for a forward difference can be written:

$$\underline{\underline{D}} = \frac{1}{\Delta x}\begin{pmatrix} -1 & 1 & 0 & 0 & 0 & \dots \\ 0 & -1 & 1 & 0 & 0 & \dots \\ 0 & 0 & -1 & 1 & 0 & \dots \\ 0 & 0 & 0 & -1 & 1 & \dots \\ \vdots & \vdots & \vdots & & \ddots & \end{pmatrix}$$

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:

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

(You can find this by using a centred first order difference formula on the first derivatives at $x + \Delta x/2$ and $x - \Delta x/2$.)

Implementation of finite differences is rather simple, but a library function is available: `np.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 of 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.

Exercises

In class

1. Create a function that implements the *forward* difference formula, taking the function $f(x)$, x and Δx as inputs and returns the approximation to the differential (in this case do *not* use `np.roll`: assume that x is a single value).

2. Test it on the sine function for one specific value of x (I chose $x = 1.0$), using `np.cos` to check the result. 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 Δx below about 10^{-5} then you will see interesting behaviour]

3. Write a function to calculate the second derivative using the centred formula. As in the previous question, test it for sine, comparing to what you know the analytic result to be.

Further work

1. Implement a function for the *centred* difference and do the same analysis you did for the forward difference in the in-class exercises. Does this behave as expected? Check that the errors in the formulae for differentials follow our analysis above (linear for forward, quadratic for centred).

2. Create an array of x values from $0 \rightarrow 2\pi - \Delta x$ with spacing $\Delta x = 2\pi/500$. Create an array holding $\cos(x)$ using `np.cos`. Now calculate the forward finite difference approximation to the differential of the array using `np.roll`, and compare to the exact result found using `np.sin`. [The shift in `np.roll` equivalent to $+\Delta x$ is an index of -1. Make sure that you understand why. Also think about why we chose the value of Δx.]

3. Write a function that combines two first derivative FD functions and compare the results to the centred second derivative you did in class (you may wish to experiment with two forward differences vs forward and back).

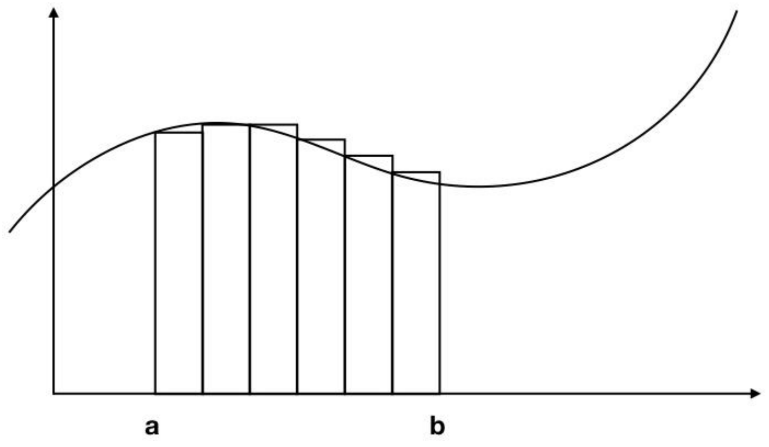
Integration

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:

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

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].



The simplest rectangle approach to numerical integration of a function between a and b

We can improve on the simplistic approach here but replacing rectangles with trapezia (the trapezium rule):

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

(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:

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

Note that in this formula we are implicitly running from *i*=0 to *i*=*N*. Of course, there are also more sophisticated approaches: Gaussian quadrature is one well-known version, which fits polynomials such that for n points it 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),

[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.

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.

Exercises

In class

1. Create a simple integration function that uses the rectangle formula. It will need to take a function, two limits and a number of points (or spacing) as arguments. You could use this interface:

```
def rectangle_int(f,a,b,N):
    """docstring: f is a Python function to be integrated"""
```

2. Use this function to 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 (copy and adapt your rectangular integration function if you like) and do the same calculation (be careful about the beginning and end points). 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. Write a Python function to integrate a two-dimensional function, $f(x, y)$, using the simple 2D extension of the rectangle rule above: you should sum over the values of the function on a 2D grid (note that `np.sum(a)` will return the sum over all the entries in the array `a`, even if 2D) and scale by the area between grid points. Your function call should look like this:

```
def integrate_2d(fun,x,y,dx,dy):
    """Docstring"""
```

3. Now test this. Define the function $f(x, y) = \cos(x)\cos(y)$ and integrate between 0 and $pi/4$ for both x and y (you should be able to work out the analytic result quite easily). Explore the effect of the grid spacing on the accuracy and the time taken. Be sure to include the end point in the integration.

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 $0 \to N$ 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.

Exercises

In-class

1. Calculate the same integrals using SciPy's routines (`trapz` and `simps`) as you did in the previous section. Compare the answers.

Further work

1. Use the library routine `quad` from `scipy.integrate` and see how accurate it is. Try adding an argument to your $x\cos(x)$ function (to change to $x\cos(\alpha x), say$), and passing it to `quad`.

Assignment

Continuing our examination of the quantum harmonic oscillator (QHO) that we introduced last week, we can in general find the energy of any wavefunction $\psi(x)$ using the formula:

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

$$E_\psi = \frac{\int \mathrm{dx}\psi^\star(x)\,\hat{H}\,\psi(x)}{\int \mathrm{dx}\psi^\star(x)\psi(x)}$$

where $\hat{H}$ is the Hamiltonian (with a hat indicating that it is an operator). If the wavefunction is correctly normalised, the denominator (bottom of the fraction) will be one and can be ignored.

For the QHO, we have a Hamiltonian given by:

$$\hat{H} = \hat{T} + \hat{V} = -\frac{1}{2}\frac{d^2}{dx^2} + \frac{1}{2}kx^2$$

where the two terms are the kinetic and potential contributions to the energy (note that we will set *k=1* here). In this assignment, you will create a wavefunction on a grid, and then calculate its second derivative using finite differences, and evaluate the kinetic and potential energies using integration on the grid.

In the first part, we will create the wavefunction and its second derivative.

1. Create a NumPy array for the x coordinate from -5 to 5 with a spacing Δx (you should choose this so that the calculations are reasonably accurate; add at least some text explaining how you chose it, and note that it's fine to run tests on accuracy and then set a value on that basis).

2. Create a NumPy array holding the wavefunction:

$$\psi(x) = (2\alpha/\pi)^{1/4}e^{-\alpha x^2}$$

3. where $\alpha = 1.0$ for the values of x you chose above.

4. Now define a function (based on the exercises in-class) to evaluate a second derivative using finite differences, and apply it to the wavefunction.

5. Plot ψ and $d^2\psi/dx^2$ against x.

In the second part, you will evaluate the energies.

1. Make a plot of the functions $(1/2)\psi(x)d^2\psi/dx^2$ and $\psi(x)(x^2/2)\psi(x)$ against x.

2. Import `integrate` from SciPy and use `integrate.simps` to evaluate the kinetic and potential energies:

$$\begin{aligned} E_{KE} &= -\frac{1}{2}\int \mathrm{dx}\psi(x)\frac{d^2\psi}{dx^2} \\ E_{PE} &= \frac{1}{2}\int \mathrm{dx}\psi(x)x^2\psi(x) \end{aligned}$$

3. Comment *briefly* on the relative sizes of the energies, and how they compare to the plots you just made.

4. (NB this is *optional*) If you want, you could explore how the relative contributions to the energies (and the plots) change with the value of α (though this will not be marked). You could check your integration by setting $\alpha = 0.5$ (the value for the ground state) and comparing the energy to the value you obtained last week.

Progress review

Once you have finished *all the material* associated with this session (both in-class and extra material), you should be able to:

- understand how grids are used to discretise space, and how they are created
- choose an appropriate finite difference approach to differentiate a function on a grid
- perform numerical integration on a function defined on a grid

You should by now be reasonably confident with the concepts and skills learned in Session 1 (NumPy arrays, their creation and manipulation; plotting with Matplotlib; loops and branching; writing and using functions).