PHAS0030: Further Practical Mathematics & Computing
Session 2: Optimisation and Root Finding

David Bowler

January 11, 2019

In this session, we will consider accuracy and precision in computing, and discuss approaches to building computational models. We will then look at methods of optimisation and searching for extrema. This will start to introduce ideas involved in writing programs.

Contents

1 Objectives 2

2 Review of Session 1 2

3 Accuracy and precision 2

3.1 Exercises 3

4 Approximating reality 4

4.1 Exercises 4

5 Finding roots of functions 6

5.1 Multi-dimensional functions 8

5.2 Exercises 9

6 Optimising multi-dimensional functions 10

6.1 Exercises 12

7 SciPy routines 13

7.1 Exercises 14

8 Progress Review 14

1 Objectives

The objectives of this session are to:

- consider accuracy and precision in computers;
- discuss how we create computer models of physics problems;
- examine approaches to root finding;
- explore optimisation of multi-dimensional non-linear functions.

2 Review of Session 1

In the first session, we started by reviewing the Python that you should already know, including:

- NumPy arrays, their creation and manipulation (including `np.zeros`, `np.ones`, `np.linspace`, `np.arange`, `np.dot`, `np.append`, `np.reshape`, `np.roll`);
- loops and flow control: **`for`**, **`while`** and **`if`**;
- functions and documentation;
- plotting with matplotlib (contrasting `plt.plot()` with `fig = plt.figure(), ax = fig.add_subplot(rcn), ax.plot()`.

These form the basis of all the programming that you will do in the rest of this course, so you should be sure that you understand them all, and are able to complete all the exercises.

3 Accuracy and precision

In experimental physics, the precision of a measurement is defined in terms of statistics, and is a measure of the random error in the experiment; it is given as the inverse of the variance of the data. The accuracy is a measure of the deviation from the "correct" value.

In computing, precision refers to the number of bits (binary digits) used to store the number. This determines both the magnitude of the number that can be stored, and the number of decimal places for a real number; these are given as $n.nnn\ldots \times 10^{exp}$. A floating point number that is stored using 8 bytes (or 64 bits)[1] uses 11 bits for the exponent, 52 for the mantissa (the digits) and one for the sign. The resulting precision gives just under sixteen decimal places, and an exponent of just under 308.

[1] This is a common choice, often referred to as double precision.

While this may seem like more than enough precision, it is important to be aware of the limits of precision. The exercises below give you some simple tests to run; for now, consider this algorithm *in your head*: what will happen ?

```
i=0
epsilon = 1.0
while 1.0+epsilon>1.0:
    i += 1
    print(i,epsilon)
    epsilon /= 2
```

You might like to compare the answer to what happens if you run it.

There are useful thoughts on floating point arithmetic in the Python tutorial: `https://docs.python.org/3.7/tutorial/floatingpoint.html`. You can also interrogate the machine precision (the limits of floating point) in Numpy using the function `np.finfo(0.1).eps` (note that 0.1 is an arbitrary choice!) while the function `np.finfo(0.1)` gives information on the resolution, limits of storage and type of a number. It is important to be aware that there are limits to precision, though they should not concern you often during the course.

3.1 *Exercises*

In class

1. Try these simple calculations: `print(0.3 - 0.2)`, `print(0.4 - 0.2)` and `print(0.3 - (0.1+0.1+0.1))`. Why do you think that we get these results?

2. Run the simple algorithm above (halving epsilon repeatedly) and be sure that you understand what happens

3. Write a function that approximates `cosine(x)` using a Taylor series expansion, taking both `x` and the number of terms in the expansion as arguments. By comparing to `np.cos` find out how the accuracy changes with number of terms (you might like to plot a graph with the result). You may use the factorial function from `math` if you wish: `from math import factorial`.

$$\cos(x) \simeq \sum_{n=0}^{N} \frac{(-1)^n}{(2n)!} x^{2n} \tag{1}$$

Further work

- We are going to approximate `cosine` by approximating the derivative of `sine`. Create a function that takes an argument, `x`, and a small change, `dx`, and approximates cosine using the definition of the derivative of sine (remember that $df/dx \simeq (f(x+dx) - f(x))/dx$). Print the difference to the actual results using `np.cos`. Test the accuracy for `dx`=1e-5, 1e-10 and 1e-15.

4 *Approximating reality*

At the heart of physics lies the art of making models of physical problems that can be solved. When using analytical approaches, these models are often simple, while computational approaches allow more complex, and often more realistic, models. We have to approximate a real-world problem with a finite computer.

We often start by breaking the continuous (time and space) into discrete chunks[2]. We will consider more of this in the next session, but the choice of the size of the chunk is important: the smaller it is, the better the description, though the more costly the computation will be.

[2] This should feel a little familiar from calculus, where we take a small change, and allow it to shrink to zero.

As well as this, we have to consider what to include in the model. We will examine a simple model of the Earth, Moon and Sun below; do we need to consider the effect of other planets on the Earth? How much of an effect does the Moon have on the Earth's orbit? And is it true the other way around?

A different example might be the electronic structure of an atom: do we need to include the effect of gravity on the electrons? The choice of what to include in a model will directly affect the complexity of the model, as well as its accuracy. It is almost always best to start as simple as possible, and then add more effects.

It is also vital to have some way to test whether or not the model is correct. Understanding the physics that has gone into the model, *and* the physics that has been omitted, is a vital part of this process. Simple order-of-magnitude estimates of what values should be found are an excellent basic check, while tests for limiting cases (as a variable goes to zero or to a large number) are also important. Ultimately, we should compare to experiment, to test our model against the real world.

4.1 *Exercises*

Much of the work in this course will involve constructing models of the world, so we only present one here. The further work will develop this.

In class The simulation is a *very* over-simplified model of the Earth-Moon-Sun system (for instance, we will assume perfectly circular orbits). We will plot the acceleration acting on the Earth and the Moon over time, building up the complexity of the simulation as we go. Start with the following set of definitions for physical constants (note that times are in seconds, which are a little inconvenient, but simple for now).

```
# Set up time
total_time = 2*365*24*60*60 # s
days = 730 # Subdivision
time = np.linspace(0,total_time,days)
# Orbits
period_ES = 365*24*60*60 # s
period_ME = 28*24*60*60  # s

# Physical constants, approximated
radius_ES = 1.5e11 # m
radius_ME = 3.8e8  # m
mass_S = 2e30      # kg
mass_E = 6e24      # kg
mass_M = 7e22      # kg
G = 6.7e-11        # m^3 /kg s^2
```

- We will assume that the Sun is at the origin, and that the Earth and Moon start on the x-axis
- We will further assume that the Earth follows a circular orbit around the Sun, so that:

$$x_{ES} = R_{ES}\cos(2\pi t/T_{ES}) \quad (2)$$
$$y_{ES} = R_{ES}\sin(2\pi t/T_{ES}) \quad (3)$$

- Create a 2D array, `pos_ES`, that holds the Earth's position at different times (use `np.array([x_ES,y_ES])` where you should use an appropriate Numpy function acting on the variable `time` defined above to replace what I have notated as `x_ES` and `y_ES`).
- Now calculate the acceleration that the Earth experiences due to the Sun's gravitational field as a 2D array:

$$\mathbf{a}_{ES} = -G\frac{M_S\mathbf{r}_{ES}}{R_{ES}^3} \quad (4)$$

  Note that $\mathbf{r}_{ES}$ is the vector position of the Earth (you may find it helpful to use `pos_ES`) while R_{ES} is the radius of the orbit defined above. Plot the two components on the same graph (you can access components easily: for instance, the x-component is `a_E[0]`), remembering to add axis labels and a title.

- Now create a new 2D array, `pos_ME`, that holds the Moon's position *relative to the Earth* (use the same approach as you did for the Earth, but remember to replace the radius and period; this will now store $\mathbf{r}_{ME}$ at different times). Add the acceleration[3] due to the Moon's gravitational field:

$$\mathbf{a}_{EM} = G\frac{M_M\mathbf{r}_{ME}}{r_{ME}^3} \quad (5)$$

  onto $\mathbf{a}_{ES}$ to get $\mathbf{a}_E$, and plot this. How much difference does the Moon make? Is this reasonable?

[3] The vectors do add correctly in this case; you will need to be more careful below when considering the Moon.

- Now plot the acceleration due to the Moon only ($\mathbf{a}_{EM}$). Note what extra information you get by plotting this, as opposed to comparing the two plots of the total acceleration.

Further work We will do the same simulation, but now for the Moon: the vectors will require a little care.

- Start by calculating the acceleration of the Moon due to the Earth's gravitational field:

$$\mathbf{a}_{ME} = -G\frac{M_E\mathbf{r}_{ME}}{R_{ME}^3} \tag{6}$$

  and plot it as before.

- We will now include the acceleration due to the Sun's gravitational field. We find $\mathbf{r}_{MS} = \mathbf{r}_{ME} + \mathbf{r}_{ES}$; note that the vector addition here is done in the same way in Numpy. The acceleration is:

$$\mathbf{a}_{MS} = -G\frac{M_S\mathbf{r}_{MS}}{|\mathbf{r}_{MS}|^3} \tag{7}$$

  You will have to calculate $|\mathbf{r}_{MS}|^3$ as a Numpy array (but remember that you can operate on entire Numpy arrays in one go: calculate $|\mathbf{r}_{MS}|$ using `np.sqrt`).

- Plot the total acceleration of the Moon, $\mathbf{a}_M = \mathbf{a}_{ME} + \mathbf{a}_{MS}$. How much effect does the Sun have compared to the Earth? Is this reasonable?

5 *Finding roots of functions*

Finding the roots of a function (solving for values of x which give $f(x) = 0$) is a key part of computational physics. As a simple example, consider the solution to the problem posed by the poet Robert Longfellow: "I shot an arrow into the air, It fell to earth, I knew not where."[4]. As physicists, we would simply write down the classical mechanics equations for the horizontal and vertical displacements of the arrow, x and z, as functions of time, and solve for the value of t that gives $z = 0$ — in other words, finding the root of the equation.

A related problem is that of optimisation: finding the maximum or minimum values of functions. Notice that if $f(x)$ is at an extremum, then $f'(x) = 0$; we will consider optimisation in the next section. All of these approaches are *iterative*: they involve the repeated application of the same basic process, something which is very easy to turn into an efficient function. We will start our consideration of root finding[5] by considering the simplest problem: a function of one dimension.

[4] The opening lines of the poem "The arrow and the song". To be fair, Longfellow was probably not thinking about classical mechanics or computational physics when he wrote the poem.

[5] This is rather different to another form of optimisation on computers: route finding...

Bisection The simplest way to find a root of a function involves knowing that a root can be found between two points, a and b; this is known as bracketing a root. We know that the values of the function at these points, $f(a)$ and $f(b)$, must have opposite signs. There is no general way to bracket a root for an arbitrary function (it is often worth plotting a function if you can) but searching downhill[6] from a until the function changes sign is often effective.

[6] Downhill here means "in the direction of decreasing magnitude", i.e. so that $|f(x)|$ becomes smaller.

Once the brackets have been found, the algorithm proceeds as follows:

1. Evaluate $f(c)$ where $c = (a + b)/2$
2. If $f(c)$ has the same sign as $f(a)$ replace a with c, otherwise replace b with c
3. Repeat until some limit or tolerance is reached[7]

[7] The question of when to stop an iteration is key in computational physics; we will discuss it below, but it will keep coming up.

The secant method We might ask whether the bisection method is the most efficient approach: why should the mid-point be the best place to evaluate the function? Indeed, this not the optimum choice. The secant method finds a better choice using an estimate of the gradient of the function (though this means that, for unpleasant functions, the root is not guaranteed to be bracketed, and the new point may be *outside* the brackets[8]).

[8] For a robust implementation, you would need to have some check that the method was not diverging.

We join $f(a)$ and $f(b)$ with a straight line, and take c as the point where this line crosses the x-axis. The notation below is a little different to the bisection method; we have two points, x_{n-1} and x_n, which are equivalent to a and b. We must specify x_0 and x_1, just as with bisection. The algorithm can be written as:

1. Calculate $\delta x_n = x_n - x_{n-1}$ and $\delta f_n = f(x_n) - f(x_{n-1})$
2. Find a new point with:

$$x_{n+1} = x_n - f(x_n)\frac{\delta x_n}{\delta f_n} \tag{8}$$

3. Repeat until tolerance has been achieved

The fraction $\delta x_n / \delta f_n$ is simply the inverse of the gradient of the staight line, used to find the new point. (Compare this formula to the Newton-Raphson formula, below, and think about how they relate.)

Using gradients: Newton-Raphson The methods we have used so far have only required the evaluation of the function itself, $f(x)$. Could we be more efficient if we included the gradient? The Newton-Raphson (NR) method does just this. It requires just one input (which will have a significant influence on which root is found if there are more than one) but doubles the number of evaluations ($f(x)$ and $f'(x)$) per step[9]. It is based on a simple Taylor expansion of the function.

[9] You will need to get used to thinking about the cost of computational approaches, as this is a key part of many implementations.

We start with an initial guess, x_0, along with $f(x_0)$ and $f'(x_0)$. The algorithm is then:

1. Evaluate $x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$
2. Evaluate $f(x_n)$ and $f'(x_n)$
3. Repeat until convergence is reached

Note that the NR method is quite capable of diverging, in some cases spectacularly. If x_0 is picked so that it is near an extremum of $f(x)$, then the step will become extremely large. Despite this, it is very powerful because it converges quickly to a root given a good initial guess. Implementations generally involve some checking to ensure that the starting point is sensible.

Look back to the secant method: this can be thought of as using a simple approximation to the gradient of the function (an approach known as *finite differences* which will be very important in the rest of the course).

It is possible to write the NR method for a multi-dimensional problem (i.e. for going beyond just one variable, x), but it turns out to be a very difficult problem for general, non-linear functions. We will consider alternatives below (first, linear problems; then approaches to non-linear problems).

Tolerances and stopping When searching for roots (or optimising a function) and iterating, you will need to specify at least one criterion for stopping the search. Typically, this is referred to as a threshold or a tolerance. For a root finder, it represents the largest value you would accept as being equivalent to zero; for an optimisation, the value that the change in a function from step to step has to drop below to indicate that the extremum has been found.

Choosing a tolerance requires balancing different factors: the accuracy that you need in the answer; the time that you are willing to spend on the computation; and, to some extent, machine precision. You will also need to have some kind of iteration counter, and a limit on the number of iterations, to ensure that some particularly nasty problem does not catch you in an infinite loop. Again, setting the maximum number of iterations requires judgement: too few, and you risk missing a correct answer; too many, and the computation will take too long.

5.1 Multi-dimensional functions

For a set of linear equations, we can write a matrix equation:

$$\underline{\underline{\mathbf{A}}}\mathbf{x} = \mathbf{b}. \tag{9}$$

If we define $\mathbf{f}(\mathbf{x}) = \underline{\underline{\mathbf{A}}}\mathbf{x} - \mathbf{b}$ as a linear, multi-dimensional function, then we can see that solutions of Eq. (9) will also give $\mathbf{f}(\mathbf{x}) = 0$.

The simplest approach to this problem is to invert the matrix $\underline{\underline{\mathbf{A}}}$. But this operation scales poorly with system size (for an $N \times N$ matrix, the cost increases proportional to N^3)[10]. A number of iterative

[10] As mentioned above, computational cost is important when considering how to solve a problem; the scaling of an approach with problem size is one thing that needs to be considered.

methods have been developed to solve this problem *without* inverting the matrix; we will encounter them, and more sophisticated variants, when dealing with partial differential equations.

The Jacobi method is particularly efficient when the matrix is diagonally dominant (has the largest values on the diagonal). We split it up: $\underline{\underline{\mathbf{A}}} = \underline{\underline{\mathbf{D}}} + \underline{\underline{\mathbf{E}}}$ where $\underline{\underline{\mathbf{D}}}$ is a diagonal matrix consisting of the diagonal entries of $\underline{\underline{\mathbf{A}}}$. This matrix can be inverted trivially, so we can write:

$$\underline{\underline{\mathbf{A}}}\mathbf{x} = \mathbf{b} \tag{10}$$

$$\left(\underline{\underline{\mathbf{D}}} + \underline{\underline{\mathbf{E}}}\right)\mathbf{x} = \mathbf{b} \tag{11}$$

$$\mathbf{x} = \underline{\underline{\mathbf{D}}}^{-1}\left(\mathbf{b} - \underline{\underline{\mathbf{E}}}\mathbf{x}\right) \tag{12}$$

Given an initial guess, $\mathbf{x}_0$, we can write an iterative procedure where the update at each step is given by:

$$\mathbf{x}_{n+1} = \underline{\underline{\mathbf{D}}}^{-1}\left(\mathbf{b} - \underline{\underline{\mathbf{E}}}\mathbf{x}_n\right) \tag{13}$$

The Gauss-Seidel method is similar in spirit: it writes $\underline{\underline{\mathbf{A}}} = \underline{\underline{\mathbf{L}}} + \underline{\underline{\mathbf{U}}}$, where $\underline{\underline{\mathbf{L}}}$ and $\underline{\underline{\mathbf{U}}}$ are lower and upper triangular matrices, respectively. If performed row-by-row, the inversion of $\underline{\underline{\mathbf{L}}}$ can be included into the update of $\mathbf{x}_n$ giving a method with the same cost as the Jacobi method (but generally better convergence). We will return to these methods when considering partial differential equations.

How well do these methods converge? That depends on the matrices that we consider. For the Jacobi method, note that at step $n+1$ we have applied the matrix product $\underline{\underline{\mathbf{D}}}^{-1}\underline{\underline{\mathbf{E}}}$ n times to x_0, so that we have $\left(\underline{\underline{\mathbf{D}}}'\underline{\underline{\mathbf{E}}}\right)^n$. For a matrix $\underline{\underline{\mathbf{B}}}$, the convergence rate can be shown to depend on the eigenvalue of $\underline{\underline{\mathbf{A}}}$ with the largest magnitude (also known as the *spectral radius* of $\underline{\underline{\mathbf{B}}}$)[11]. The larger this value, the slower the convergence.

[11] This is defined as $\rho(\underline{\underline{\mathbf{B}}}) = \max|\lambda_i|$ with λ_i the eigenvalues of $\underline{\underline{\mathbf{B}}}$.

In the next section, we will consider how optimisation can be used for *non-linear* problems, and how this can be mapped back onto root finding.

5.2 *Exercises*

In class We will be seeking the roots of the cubic function:

$$f(x) = x^3 - 2x^2 - x + 2 \tag{14}$$

using the different methods described above.

1. Write a function to calculate $f(x)$ (choose a sensible name)
2. Plot $f(x)$ between -2 and 3, to get a sense of its behaviour (you may find the command `plt.axhline` useful to add a line at $x = 0$).
3. Write a simple loop to implement the bisection method for $f(x)$; choose brackets that enclose one root (by inspection from the plot above). You should specify a tolerance to end the loop (remember the function `abs`), and keep track of how many iterations the loop requires.

4. Feel free to experiment with the effect of different brackets: what happens if more than one root is enclosed?

5. Now write a function to implement the secant method. The function should take a function as one of the inputs (so that it is a general implementation)[12], along with brackets and a tolerance. The main problem is book-keeping (updating the terms in the secant). Return the root *and* the number of iterations taken.

6. Call your function, and compare the result to bisection.

[12] This is simple to do: simply make one of the inputs `fun`, say, and then pass the name of your cubic function when you call the secant function.

Further work

1. Implement the Newton-Raphson method generally in a function: you will need to pass inputs of the function and its differential, a starting guess and a tolerance. As with the secant method, return the root and the number of iterations

2. How would you put checks in place to ensure that you caught any divergence in the NR method? Implement one of them.

3. If you are feeling adventurous, you could *combine* the bisection and NR methods: start with bisection to a loose tolerance, and then finish with NR.

6 *Optimising multi-dimensional functions*

Finding the minimum of a function, often a function of a multi-dimensional vector, is one of the key operations in computational physics: we frequently seek the lowest energy or ground state of a system. There is, however, no way to guarantee that we will find the absolute minimum (rather than a local minimum) of a function. In this section we will discuss approaches to optimisation.

If we assume that a function of a multi-dimensional vector $\mathbf{x}$ is quadratic[13] then we can write:

$$f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T\underline{\underline{\mathbf{A}}}\mathbf{x} - \mathbf{b}^T\mathbf{x} + c \tag{15}$$

[13] This is often a good approximation near a minimum; even away from a minimum this asumption is often effective.

where the matrix $\underline{\underline{\mathbf{A}}}$ is known as the Hessian: it is the curvature or second derivative of $f(\mathbf{x})$. Notice also that, so long as $\underline{\underline{\mathbf{A}}}$ is positive definite[14], we can also write:

$$\nabla_{\mathbf{x}} f(\mathbf{x}) = \mathbf{f}'(\mathbf{x}) = \underline{\underline{\mathbf{A}}}\mathbf{x} - \mathbf{b} \tag{16}$$

[14] For a Hermitian matrix, this is equivalent to having all eigenvalues greater than zero.

so that a value of $\mathbf{x}$ that gives an extremum of $f(\mathbf{x})$ will give $\mathbf{f}'(\mathbf{x}) = 0$.

In the spirit of the iterative root finding methods we discussed earlier, we can create iterative approaches to optimisation. The step which is repeated is to choose a direction (a search direction) and minimise the energy of the function along that direction, repeating

until the energy converges. We write $\mathbf{x}_{n+1} = \mathbf{x}_n + \lambda_n \mathbf{h}_n$ and search for the value of λ_n that gives the energy minimum in that direction. If the function is linear then there are exact results for λ; if it is non-linear, then we need to perform a *line search*, which is a further iterative process.

At its simplest, a line search can be written as follows:

1. Store $f(\mathbf{x}_n)$
2. Choose a starting value of λ and evaluate $f(\mathbf{x}_n + \lambda_n \mathbf{h}_n)$
3. Increase λ and repeat until $f(\mathbf{x}_n + \lambda_n \mathbf{h}_n) > f(\mathbf{x}_n)$ (at which point you will have bracketed *the minimum*)
4. Refine λ using an approach such as bisection or by fitting a parabola to three points

The simplest approach to function minimisation, known as steepest descents, defines the search direction as the local downhill direction:

$$\mathbf{h}_n = \mathbf{g}_n = -\nabla f(\mathbf{x}_n) \tag{17}$$

We then perform a line search for that direction. Once the minimum has been located, the gradient is evaluated at $\mathbf{x}_{n+1}$ and the process repeats.

However, this is not very efficient: at each step, the search direction takes no account of the previous search directions, and the efficiency of the method depends strongly on the starting point chosen. This is illustrated in Fig. 1, where steepest descents takes many more steps than is reasonable to find the lowest point in a 2D landscape.

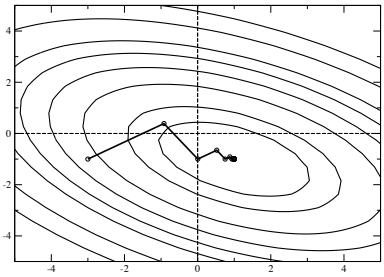

Figure 1: A 2D landscape with the minimum at (-1,1). If steepest descents is started as shown at (-3,-1), then 12 steps are required to come within 0.1% of the minimum.

It is possible to create a method with the same computational cost and simplicity as steepest descents, but which ensures that each new search direction does not undo the optimisation of the previous step. This is the method of *conjugate gradients*. Using this method described below, the search of the landscape in Fig. 1 would take two steps.

Simply put, in this context, a conjugate direction is one that will not affect the minimisation of the previous line search; this can be defined as $\mathbf{x_n}^T \underline{\underline{\mathbf{A}}} \mathbf{x}_{n+1} = 0$. Of course, we don't know what the matrix $\underline{\underline{\mathbf{A}}}$ is, but it can be shown that the following iteration constructs conjugate directions *without* knowing $\underline{\underline{\mathbf{A}}}$. We define $\mathbf{h}_0 = \mathbf{g}_0$, and $\mathbf{g}_n = -\nabla f(\mathbf{x}_n)$ and iterate:

1. Calculate $\mathbf{g}_{n+1}$ and evaluate $\gamma_n = \mathbf{g}_{n+1} \cdot \mathbf{g}_{n+1} / \mathbf{g}_n \cdot \mathbf{g}_n$
2. Then:
$$\mathbf{h}_{n+1} = \mathbf{g}_{n+1} + \gamma_n \mathbf{h}_n \tag{18}$$
3. It can be shown that $\mathbf{g}_n \cdot \mathbf{g}_{n+1} = 0$ and $\mathbf{h}_n^T \underline{\underline{\mathbf{A}}} \mathbf{h}_{n+1} = 0$

Note that the directions are only exactly conjugate if the function being minimised is quadratic; the method can be applied effectively even if it isn't, but the directions will not be perfectly conjugate. It is common practice to reset the search direction to the

local downhill direction from time to time. You can find a very full discussion of steepest descents and conjugate gradients here: `http://www.cs.cmu.edu/~jrs/jrspapers.html#cg`.

The efficiency of conjugate gradients, or any minimisation scheme, is strongly dependent on the shape of the energy landscape. In simple 2D terms a valley with very different curvatures (very steep in one direction and very shallow in the other) is hard to optimise. More formally this can be characterised by the spectral conditioning of the curvature: $\kappa = |\lambda_{\max}| \,/\, |\lambda_{\min}|$. If $\kappa \gg 1$ then the convergence is likely to be slow (the matrix is known as *ill-conditioned*). There is a technique called pre-conditioning which seeks to reduce κ by transforming the curvature matrix, but we do not have time to consider this in detail.

6.1 Exercises

We are going to plot a 2D function using `plt.imshow`, exploring two possible ways to do this. In the further work you will explore steepest descents and conjugate gradients.

In class

1. Write a Python function that evaluates the 2D function:

$$f(x,y) = \frac{3}{2}x^2 + 2xy + 3y^2 - x + 4y \tag{19}$$

2. First, create 1D arrays for x and y between -5 and 5 (don't use more than 50 points in the first place). Create a 2D array to store the results of the function. Then loop over both x and y (you will need nested loops, and might like to use `enumerate`) and evaluate the function at all points, storing the result in the appropriate place in your 2D array; consider the fragment below.

```
for i, x in enumerate(x_arr):
    for j, y in enumerate(y_arr):
        store_array[i,j] = your_function(x,y)
```

3. Plot using `plt.imshow`. You might want to pass the extent to `plt.imshow` (`extent=(xmin,xmax,ymin,ymax)` and set the origin to be lower left (`origin='lower'`).

4. Another approach is pass the full set of points into the Python function and get back a full, 2D array. However, to this we need to create 2D x and y arrays (if you think about it, we are making a matrix of function values, in effect, and so we need one matrix containing the values for x at every point in space and another for y; the rows for x will all be the same while the columns for y will be the same). We could do this ourselves, but the function `np.meshgrid` is designed to take two 1D arrays and return the appropriate 2D arrays:

```
x2D, y2D = np.meshgrid(x_arr, y_arr,indexing='ij')
```

Use the fragment above to create appropriate arrays, and pass them directly to your function (which will now return the full 2D array that you want to plot). Try to get a sense of which of these approaches is faster (if necessary increase the size of your arrays, but don't make them too large !).

Further work

1. Write a function that returns the vector gradient of $f(x, y)$ from above (using the components $\partial f/\partial x$ and $\partial f/\partial y$)
2. Write a function that implements a simple line search: given a function, a starting point, $\mathbf{x}$, and a search direction, $\mathbf{h}(\mathbf{x})$, you should test the value of $f(\mathbf{x} + \lambda\mathbf{h})$ for increasing values of λ until the value goes up (this will indicate that you have bracketed the minimum). You only need store the current and new values of the function. Use a simple bisection routine to refine the minimum after bracketing.

7 *SciPy routines*

There are often efficient implementations of well-known algorithms, and this is true for root finding and optimisation. We have started by describing how the algorithms work, and explored implementations, as it is vital not to treat these complex problems as black boxes: there is rarely any absolute guarantee of convergence, however sophisticated an approach, and you will need to know the limitations of the methods that you are using.

We need to introduce a new Python module, SciPy, to explore the implementations of optimization methods. The convention for importing is a little different to NumPy: we import the *sub-module* directly from SciPy. For this section, the relevant sub-module is `optimize`, so you would need to put:

```
from scipy import optimize
```

at the start of your code or notebook. If you explore the manual pages (`https://docs.scipy.org/doc/scipy/reference/optimize.html`) then you will see that there are many different approaches to optimization.

There are various routines of direct interest:

- Bisection: `optimize.bisect(f,a,b)` which takes optional arguments[15] `xtol` (setting the tolerance) and `maxiter` (setting the maximum number of iterations). It returns the value of the root.
- Newton-Raphson: `optimize.newton(f,x0)` with the optional arguments `fprime` (the derivative of the function—without this it will default to the secant method), `tol` and `maxiter`.

[15] For a Python function, you specify an optional argument by giving its name and assigning it a value; for example `optimize.bisect(f,a,b,xtol=1e-6`.

- Conjugate gradients: `optimize.minimize(fun, x0, method='CG', jac=d_fun)`, where we pass the function as `fun` and its gradient as `d_fun` and `x0` is a starting point. Notice that `fun` should accept an array which is the same size as `x0`, as should `d_fun`, which should return an array of this size. The function returns a compound variable (an object) which will require a little unpacking: if you stored the return value as `opt_result` then the minimum will be the array `opt_result.x` and the number of iterations required will be `opt_result.nit`.

7.1 Exercises

In class

1. Test the SciPy `bisect` method against your implementation above.

Further work

1. Test the SciPy `newton` method against your implementation above.
2. Optimise the function from the exercises in Section 6.1 using conjugate gradients from SciPy. See the notes on the returned object above.

8 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 the basic ideas of precision and representation of floating point numbers in computers
- begin creating models of physical problems in computer code
- choose an appropriate method to find the roots of a function
- understand the basic concepts of optimisation of multi-dimensional functions
- use SciPy optimization and root finding functions

You should also check that you understand all the concepts and skills practised in Session 1.