Chapter 1

Introduction

This text summarises a number of core ideas relevant to Computational Engineering and Scientific Computing using Python. The emphasis is on introducing some basic Python (programming) concepts that are relevant for numerical algorithms. The later chapters touch upon numerical libraries such as `numpy` and `scipy` each of which deserves much more space than provided here. We aim to enable the reader to learn independently how to use other functionality of these libraries using the available documentation (online and through the packages itself).

1.1 Computational Modelling

1.1.1 Introduction

Increasingly, processes and systems are researched or developed through computer simulations: new aircraft prototypes such as for the recent A380 are first designed and tested virtually through computer simulations. With the ever increasing computational power available through supercomputers, clusters of computers and even desktop and laptop machines, this trend is likely to continue.

Computer simulations are routinely used in fundamental research to help understand experimental measurements, and to replace – for example – growth and fabrication of expensive samples/experiments where possible. In an industrial context, product and device design can often be done much more cost effectively if carried out virtually through simulation rather than through building and testing prototypes. This is in particular so in areas where samples are expensive such as nanoscience (where it is expensive to create small things) and aerospace industry (where it is expensive to build large things). There are also situations where certain experiments can only be carried out virtually (ranging from astrophysics to study of effects of large scale nuclear or chemical accidents). Computational modelling, including use of computational tools to post-process, analyse and visualise data, has been used in engineering, physics and chemistry for many decades but is becoming more important due to the cheap availability of computational resources. Computational Modelling is also starting to play a more important role in studies of biological systems, the economy, archeology, medicine, health care, and many other domains.

1.1.2 Computational Modelling

To study a process with a computer simulation we distinguish two steps: the first one is to develop a *model* of the real system. When studying the motion of a small object, such as a penny, say, under the influence of gravity, we may be able to ignore friction of air: our model — which might only consider the gravitational force and the penny's inertia, i.e. $a(t) = F/m = -9.81\text{m/s}^2$ — is an approximation of the real system. The model will normally allow us to express the behaviour of the system (in

some approximated form) through mathematical equations, which often involve ordinary differential equations (ODEs) or partial differential equatons (PDEs).

In the natural sciences such as physics, chemistry and related engineering, it is often not so difficult to find a suitable model, although the resulting equations tend to be very difficult to solve, and can in most cases not be solved analytically at all.

On the other hand, in subjects that are not as well described through a mathematical framework and depend on behaviour of objects whose actions are impossible to predict deterministically (such as humans), it is much more difficult to find a good model to describe reality. As a rule of thumb, in these disciplines the resulting equations are easier to solve, but they are harder to find and the validity of a model needs to be questioned much more. Typical examples are attempts to simulate the economy, the use of global resources, the behaviour of a panicking crowd, etc.

So far, we have just discussed the development of *models* to describe reality, and using these models does not necessarily involve any computers or numerical work at all. In fact, if a model's equation can be solved analytically, then one should do this and write down the solution to the equation.

In practice, hardly any model equations of systems of interest can be solved analytically, and this is where the computer comes in: using numerical methods, we can at least study the model *for a particular set of boundary conditions*. For the example considered above, we may not be able to easily see from a numerical solution that the penny's velocity under the influence of gravity will change linearly with time (which we can read easily from the analytical solution that is available for this simple system: $v(t) = t \cdot 9.81\mathrm{m/s}^2 + v_0$).

The numerical solution that can be computed using a computer would consist of data that shows how the velocity changes over time for a particular initial velocity v_0 (v_0 is a boundary condition here). The computer program would report a long lists of two numbers keeping the (i) value of time t_i for which a particular (ii) value of the velocity v_i has been computed. By plotting all v_i against t_i, or by fitting a curve through the data, we may be able to understand the trend from the data (which we can just see from the analytical solution of course).

It is clearly desirable to find an analytical solutions wherever possible but the number of problems where this is possible is small. Usually, the obtaining numerical result of a computer simulation is very useful (despite the shortcomings of the numerical results in comparison to an analytical expression) because it is the only possible way to study the system at all.

The name *computational modelling* derives from the two steps: (i) *modelling*, i.e. finding a model description of a real system, and (ii) solving the resulting model equations using *computational* methods because this is the only way the equations can be solved at all.

1.1.3 Programming to support computational modelling

A large number of packages exist that provide computational modelling capabilities. If these satisfy the research or design needs, and any data processing and visualisation is appropriately supported through existing tools, one can carry out computational modelling studies without any deeper programming knowledge.

In a research environment – both in academia and research on new products/ideas/... in industry – one often reaches a point where existing packages will not be able to perform a required simulation task, or where more can be learned from analysing existing data in news ways etc.

At that point, programming skills are required. It is also generally useful to have a broad understanding of the building blocks of software and basic ideas of software engineering as we use more and more devices that are software-controlled.

It is often forgotten that there is nothing the computer can do that we as humans cannot do. The computer can do it much faster, though, and also with making far fewer mistakes. There is thus no magic in computations a computer carries out: they could have been done by humans, and – in fact – were for many years (see for example Wikipedia entry on Human Computer).

Understanding how to build a computer simulation comes roughly down to: (i) finding the model (often this means finding the right equations), (ii) knowing how to solve these equations numerically, (ii) to implement the methods to compute these solutions (this is the programming bit).

1.2 Why Python for scientific computing?

The design focus on the Python language is on productivity and code readability, for example through:

- Interactive python console
- Very clear, readable syntax through whitespace indentation
- Strong introspection capabilities
- Full modularity, supporting hierarchical packages
- Exception-based error handling
- Dynamic data types & automatic memory management

As Python is an interpreted language, and it runs many times slower than compiled code, one might ask why anybody should consider such a 'slow' language for computer simulations?

There are two replies to this criticism:

1. *Implementation time versus execution time*: It is not the execution time alone that contributes to the cost of a computational project: one also needs to consider the cost of the development and maintenance work.

   In the early days of scientific computing (say in the 1960/70/80), compute time was so expensive that it made perfect sense to invest many person months of a programmer's time to improve the performance of a calculation by a few percent.

   Nowadays, however, the CPU cycles have become much cheaper than the programmer's time. For research codes which often run only a small number of times (before the researchers move on to the next problem), it may be economic to accept that the code runs only at 25% of the expected possible speed if this saves, say, a month of a researcher's (or programmers) time. For example: if the execution time of the piece of code is 10 hours, and one can predict that it will run about 100 times, then the total execution time is approximately 1000 hours. It would be great if this could be reduced to 25% and one could save 750 (CPU) hours. On the other hand, is an extra wait (about a month) and the cost of 750 CPU hours worth investing one month of a person's time [who could do something else while the calculation is running]? Often, the answer is not.

   Code readability & maintenance - short code, fewer bugs: A related issue is that a research code is not only used for one project, but carries on to be used again and again, evolves, grows, bifurcates etc. In this case, it is often justified to invest more time to make the code fast. At the same time, a significant amount of programmer time will go into (i) introducing the required changes, (ii) testing them even before work on speed optimisation of the changed version can start. To be able to maintain, extend and modify a code in often unforeseen ways, it can only be helpful to use a language that is easy to read and of great expressive power.

2. *Well-written Python code can be very fast* if time critical parts in executed through compiled language.

   Typically, less than 5% percent of the code base of a simulation project need more than 95% of the execution time. As long as these calculations are done very efficiently, one doesn't need to worry about all other parts of the code as the overall time their execution takes is insignificant.

   The compute intense part of the program should to be tuned to reach optimal performance. Python offers a number of options.

   - For example, the `numpy` Python extension provides a Python interface to the compiled and efficient LAPACK libraries that are the quasi-standard in numerical linear algebra. If the problems under study can be formulated such that eventually large systems of algebraic equations have to be solved, or eigenvalues computed, etc, then the compiled code in the LAPACK library can be used (through the Python-numpy package). At this stage, the calculations are carried out with the same performance of Fortran/C as it is essentially Fortran/C code that is used. Matlab, by the way, exploits exactly this: the Matlab scripting language is very slow (about 10 time slower than Python), but Matlab gains its power from delegating the matix operation to the compiled LAPACK libraries.
   - Existing numerical C/Fortran libraries can be interfaced to be usable from within Python (using for example Swig, Boost.Python and Cython).
   - Python can be extended through compiled languages if the computationally demanding part of the problem is algorithmically non-standard and no existing libraries can be used.

     Commonly used are C, Fortran and C++ to implement fast extensions.
   - We list some tools that are used to use compiled code from Python:
     - ▷ The `scipy.weave` extension is useful if just a short expression needs to be expressed in C.
     - ▷ The Cython interface is growing in popularity to (i) semi-automatically declare variable types in Python code, to translate that code to C (automatically) and to then use the compiled C code from Python. Cython is also used to quickly wrap an existing C library with an interface so the C library can be used from Python.
     - ▷ Boost.Python is specialised for wrapping C++ code in Python.

The conclusion is that Python is "fast enough" for most computational tasks, and that its user friendly high-level language often makes up for reduced speed in comparison to compiled lower-level languages. Combining Python with tailor-written compiled code for the performance critical parts of the code, results in virtually optimal speed in most cases.

1.2.1 Optimisation strategies

We generally understand reduction of execution time when discussing "code optimisation" in the context of computational modelling, and we essentially like to carry out the required calculations as fast as possible. (Sometimes we need to reduce the amount of RAM, the amount of data input output to disk or the network.) At the same time, we need to make sure that we do not invest inappropriate amounts of programming time to achieve this speed up: as always there needs to be a balance between the programmers' time and the improvement we can gain from this.

1.3.2 Python tutor mailing list

There is also a Python tutor mailing list (http://mail.python.org/mailman/listinfo/tutor) where beginners are welcome to ask questions regarding Python. Both using the archives and posting your own queries (or in fact helping others) may help with understanding the language. Use the normal mailing list etiquette (i.e. be polite, concise, etc). You may want to read http://www.catb.org/esr/faqs/smart-questions.html for some guidance on how to ask questions on mailing lists.

1.4 Python version

There are two version of the Python language out there: Python 2.x and Python 3.x. They are (slightly) different — the changes in Python 3.x were introduced to address shortcomings in the design of the language that were identified since Python's inception. A decision has been made that some incompatibilty should be accepted to achieve the higher goal of a better language for the future.

For scientific computation, it is crucial to make use of numerical libraries such as numpy, scipy and the plotting package matplotlib.

All of these are available for Python 2.x, and increasingly they are also available for Python 3 (in fact the libraries above have all been ported by now). As Python 2.x is still the default Python on many system and there are a fair number of research codes out there based on Python 2, we will use Python 2.x in this book.

However, we will write code that is as much as possible in the Python 3 style (and understood by Python 2). The most prominent example is that in Python 2.x, the `print` command is special where as in Python 3 it is an ordinary function. For example, in Python 2.7, we can write

```
print "Hello World"
```

where as in Python 3, this would cause a SyntaxError. The right way to use `print` in Python 3 would be as a function, i.e.

```
print("Hello World")
```

See also section 5.1.5 on page 54 for further details.

Fortunately, the function notation (i.e. with the parantheses) is also allowed in Python 2.7, so we choose this notation in our examples and thus they will execute in Python 2.7 and Python 3.x. (There are other differences.)

The transition of all actively maintained codes from Python 2 to Python 3 is likely to take at least another 5 years, maybe 10. It could also be that Python 2.7 will remain longer actively used – this is hard to predict at the moment.

1.5 This document

This document has been typeset with the `\hyperref` package. This means that all entries in the table of contents, figure numbers, page numbers and URLs should appear as clickable hyperlinks if your pdf browser supports this. You may want to make use of this as often there are URLs provided that provide further information/documentation.

1.6 Your feedback

is desired. If you find anything wrong in this text, or have suggestions how to change or extend it, please feel free to contact Hans at `fangohr@soton.ac.uk`.