1 Introduction

Beginnings are hard.
Chaim Potok

Nothing is more expensive than a start.
Friedrich Nietzsche

This book is really two books. There is a rather traditional paper one with a related Web site, as well as an eBook version containing a variety of digital features best experienced on a computer. Yet even if you are reading from paper, you can still avail yourself of many of digital features, including video-based lecture modules, via the book's Web sites: http://physics.oregonstate.edu/~rubin/Books/CPbook/eBook/Lectures/ and www.wiley.com/WileyCDA.
We start this chapter with a description of how computational physics (CP) fits into physics and into the broader field of computational science. We then describe the subjects we are to cover, and present lists of all the problems in the text and in which area of physics they can be used as computational examples. The chapter finally gets down to business by discussing the Python language, some of the many packages that are available for Python, and some detailed examples of the use of visualization and symbolic manipulation packages.

1.1 Computational Physics and Computational Science

This book presents computational physics (CP) as a subfield of computational science. This implies that CP is a multidisciplinary subject that combines aspects of physics, applied mathematics, and computer science (CS) (Figure 1.1a), with the aim of solving realistic and ever-changing physics problems. Other computational sciences replace physics with their discipline, such as biology, chemistry, engineering, and so on. Although related, computational science is *not* part of computer science. CS studies computing for its own intrinsic interest and develops the hardware and software tools that computational scientists use. Likewise, applied mathematics develops and studies the algorithms that computational scientists use. As much as we also find math and CS interesting for their own sakes,

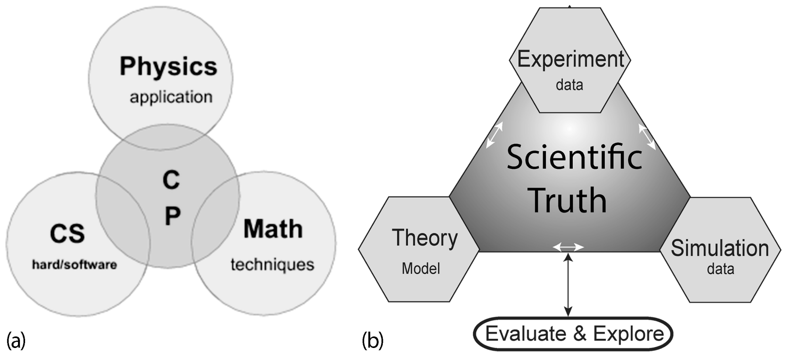


Figure 1.1 (a) A representation of the multidisciplinary nature of computational physics as an overlap of physics, applied mathematics and computer science, and as a bridge among them. (b) Simulation has been added to experiment and theory as a basic approach in the search for scientific truth. Although this book focuses on simulation, we present it as part of the scientific process.

our focus is on helping the reader do better physics for which you need to understand the CS and math well enough to solve your problems correctly, but not to become an expert programmer.

As CP has matured, we have come to realize that it is more than the overlap of physics, computer science, and mathematics. It is also a bridge among them (the central region in Figure 1.1a) containing core elements of it own, such as computational tools and methods. To us, CP's commonality of tools and its problem-solving mindset draws it toward the other computational sciences and away from the subspecialization found in so much of physics. In order to emphasize our computational science focus, to the extent possible, we present the subjects in this book in the form of a *Problem* to solve, with the components that constitute the solution separated according to the scientific problem-solving paradigm (Figure 1.1b). In recent times, this type of problem-solving approach, which can be traced back to the post-World War II research techniques developed at US national laboratories, has been applied to science education where it is called something like *computational scientific thinking*. This is clearly related to what the computer scientists more recently have come to call *Computational Thinking*, but the former is less discipline specific. Our computational scientific thinking is a hands-on, inquiry-based project approach in which there is problem analysis, a theoretical foundation that considers computability and appropriate modeling, algorithmic thinking and development, debugging, and an assessment that leads back to the original problem.

Traditionally, physics utilizes both experimental and theoretical approaches to discover scientific truth. Being able to transform a theory into an algorithm requires significant theoretical insight, detailed physical and mathematical understanding, and a mastery of the art of programming. The actual debugging, testing, and organization of scientific programs are analogous to experimentation, with the numerical simulations of nature being virtual experiments. The synthesis of numbers into generalizations, predictions, and conclusions requires the insight and intuition common to both experimental and theoretical science. In fact, the use of computation and simulation has now become so prevalent and essential a part of the scientific process that many people believe that the scientific paradigm has been extended to include simulation as an additional pillar (Figure 1.1b). Nevertheless, as a science, CP must hold experiment supreme, regardless of the beauty of the mathematics.

1.2 This Book's Subjects

This book starts with a discussion of Python as a computing environment and then discusses some basic computational topics. A simple review of computing hardware is put off until Chapter 10, although it also fits logically at the beginning of a course. We include some physics applications in the first third of this book, by put off most CP until the latter two-thirds of the book.

This text have been written to be accessible to upper division undergraduates, although many graduate students without a CP background might also benefit, even from the more elementary topics. We cover both ordinary and partial differential equation (PDE) applications, as well as problems using linear algebra, for which we recommend the established subroutine libraries. Some intermediate-level analysis tools such as discrete Fourier transforms, wavelet analysis, and singular value/principal component decompositions, often poorly understood by physics students, are also covered (and recommended). We also present various topics in fluid dynamics including shock and soliton physics, which in our experience physics students often do not see otherwise. Some more advanced topics include integral equations for both the bound state and (singular) scattering problem in quantum mechanics, as well as Feynman path integrations.

A traditional way to view the materials in this text is in terms of its use in courses. In our classes (CPUG, 2009), we have used approximately the first third of the text, with its emphasis on computing tools, for a course called *Scientific Computing* that is taken after students have acquired familiarity with some compiled language. Typical topics covered in this one-quarter course are given in Table 1.1, although we have used others as well. The latter two-thirds of the text, with its greater emphasis on physics, has typically been used for a two-quarter (20-week) course in CP. Typical topics covered for each quarter are given in Table 1.2. What with many of the topics being research level, these materials can easily be used for a full year's course or for extended research projects.

The text also uses various symbols and fonts to help clarify the type of material being dealt with. These include:

⊙	Optional material
`Monospace font`	Words as they would appear on a computer screen
Vertical gray line	Note to reader at the beginning of a chapter saying

Table 1.1 Topics for one-quarter (10 Weeks) scientific computing course.

Week	Topics	Chapter	Week	Topics	Chapter
1	OS tools, limits	1, (10)	6	Matrices, N-D search	6
2	Visualization, Errors	1, 3	7	Data fitting	7
3	Monte Carlo,	4, 4	8	ODE oscillations	8
4	Integration, visualization	5, (1)	9	ODE eigenvalues	8
5	Derivatives, searching	5, 7	10	Hardware basics	10

Table 1.2 Topics for two-quarters (20 Weeks) computational physics course.

	Computational Physics I			Computational Physics II	
Week	Topics	Chapter	Week	Topics	Chapter
1	Nonlinear ODEs	8, 9	1	Ising model, Metropolis	17
2	Chaotic scattering	9	2	Molecular dynamics	18
3	Fourier analysis, filters	12	3	Project completions	—
4	Wavelet analysis	13	4	Laplace and Poisson PDEs	19
5	Nonlinear maps	14	5	Heat PDE	19
6	Chaotic/double pendulum	15	6	Waves, catenary, friction	21
7	Project completion	15	7	Shocks and solitons	24
8	Fractals, growth	16	8	Fluid dynamics	25
9	Parallel computing, MPI	10, 11	9	Quantum integral equations	26
10	More parallel computing	10, 11	10	Feynman path integration	17

1.3 This Book's Problems

For this book to contribute to a successful learning experience, we assume that the reader will work through what we call the *Problem* at the beginning of each discussion. This entails studying the text, writing, debugging, and running programs, visualizing the results, and then expressing in words what has been performed and what can be concluded. As part of this approach, we suggest that the learner write up a mini lab report for each problem containing sections on

Equations solved	Numerical method	Code listing
Visualization	Discussion	Critique

Although we recognize that programming is a valuable skill for scientists, we also know that it is incredibly exacting and time-consuming. In order to lighten the workload, we provide "bare bones" programs. We recommend that these be used as guides for the reader's own programs, or tested and extended to solve the problem at hand. In any case, they should be understood as part of the text.

While we think it is best to take a course, or several courses, in CP, we recognize that this is not always possible and some instructors may only be able to include some CP examples in their traditional courses. To assist in this latter endeavor, in this section we list the location of each problem distributed throughout the text and the subject area of each problem. Of course this is not really possible with a multidisciplinary subject like CP, and so there is an overlap. The code used in the table for different subjects is: QM = quantum mechanics or modern physics, CM = classical mechanics, NL = nonlinear dynamics, EM = electricity and magnetism, SP = statistical physics, MM = mathematical methods as well as tools, FD = fluid dynamics, CS = computing fundamentals, Th = thermal physics, and BI = biology. As you can see from the tables, there are many problems and exercises, which reflects our view that you learn computing best by doing it, and that many problems cover more than one subject.

Problems and exercises in computational basics

Subject	Section	Subject	Section	Subject	Section
MM, CS	1.6	CS	2.2.2	CS	2.2.2
CS	2.4.3	CS	2.4.5	CS	2.5.2
CS	3.1.2	CS	3.2	CS	3.2.2
CS	3.3	CS	3.3.1	CS	4.2.2
MM, CS	6.6	CS	10.13.1	CS	10.14.1
CS	11.3.1	CS	11.1.2	CS	11.2.1

Problems and exercises in thermal physics and statistical physics

Subject	Section	Subject	Section	Subject	Section
SP, MM	4.3	SP, MM	4.5	QM, SP	4.6
Th, SP	7.4	Th, SP	7.4.1	NL, SP	16.3.3
NL, SP	16.4.1	NL, SP	16.7.1	NL, SP	16.7.1
NL, SP	16.8	NL, SP	16.11	SP, QM	17.4.1
SP, QM	17.4.2	SP, QM	17.6.2	Th, MM	20.2.4
Th, MM	20.3	TH, MM	20.4.2	TH, MM	20.1
TH, MM	17.1	SP	16.2	SP, BI	16.3
SP	16.4	SP, MM	16.5	SP	16.6
SP	16.7				

Problems and exercises in electricity and magnetism

Subject	Section	Subject	Section	Subject	Section
EM, MM	19.6	EM, MM	19.7	EM, MM	19.8
EM, MM	19.9	EM, MM	23.2	EM, MM	23.5
EM, MM	23.5.1	EM, MM	23.6.6	EM, MM	22.7.2
EM, MM	22.10	EM, MM	19.2		

Problems and exercises in quantum mechanics

Subject	Section	Subject	Section	Subject	Section
QM, SP	4.6	QM, MM	7.1	QM, MM	7.2.1
QM, MM	7.3.2	QM, MM	9.1	QM, MM	9.2
QM, MM	9.2.1	QM, MM	9.3	QM	13.6.3
QM, MM	17.7	QM, MM	26.1	QM, MM	26.3
QM, MM	22.1				

Problems and exercises in classical mechanics and nonlinear dynamics

Subject	Section	Subject	Section	Subject	Section
CM, NL	5.16	CM	6.1	CM, NL	8.1
CM, NL	8.7.1	CM, NL	8.8	CM, NL	8.9
CM, NL	8.10	CM, NL	9.4	CM, NL	9.4.3
CM	9.5	CM	9.7	CM	9.7
NL, FD	9.7	CM	9.7	CM, MM	6.6.2
CM, MM	6.6.1	CM, NL	12.1	BI, NL	14.3
CM, MM	6.6.1	BI, NL	14.4	BI, NL	14.5.2
BI, NL	14.5.3	BI, NL	14.10	BI, NL	14.11.1
BI, NL	14.11.4	BI, NL	14.11.5	CM, NL	15.1.3
CM, NL	15.1	NL, BI	14.1	NL, BI	14.9
CM, NL	15.2.2	CM, NL	15.3	CM, NL	15.4
CM, NL	15.5	CM, NL	15.6	CM, NL	15.7
CM, NL	15.7	NL, MM	16.2.1	NL, MM	16.3.3
NL, MM	16.4.1	NL, MM	16.5.3	NL, MM	16.7.1
NL, MM	16.7.1	NL, MM	16.8	NL, MM	16.11
CM, MM	21.2.4	CM, MM	21.3	CM, MM	21.4.3
CM, MM	24.6	CM, MM	21.1	CM, MM	21.5

Problems and exercises in fluid dynamics

Subject	Section	Subject	Section	Subject	Section
NL, FD	9.7	FD, MM	24.3.2	FD, MM	24.5.3
FD, MM	24.5.4	FD, MM	25.1	FD, MM	25.2.3
FD, MM	25.4.4	FD, MM	25.4.5		

Problems and exercises in mathematical methods and computational tools

Subject	Section	Subject	Section	Subject	Section
MM, CS	1.6	MM, SP	4.3	SP, MM	4.3.2
BI, MM	4.4	MM, SP	4.5	MM	5.12.3
MM	5.16	MM	5.17.2	MM	5.5
MM	5.5	QM, MM	7.1	QM, MM	7.2.1
QM, MM	7.3.2	MM, QM	9.1	QM, MM	9.2
QM, MM	9.2.1	QM, MM	9.3	CM, NL	9.4
MM, CS	6.6	CM, MM	6.6.2	CM, MM	6.6.1
MM	7.5.1	MM	7.5.2.1	MM	7.8
MM	7.8.1	MM	7.8.2	MM	12.3
MM	12.5.3	MM	12.7.1	MM	12.11
MM	13.3.1	MM	13.5.2	MM	13.6.3
CM, MM	15.5	NL, MM	16.2.1	NL, MM	16.3.3
NL, MM	16.4.1	NL, MM	16.5.3	NL, MM	16.7.1
NL, MM	16.7.1	NL, MM	16.8	NL, MM	16.11
Th, MM	20.2.4	Th, MM	20.3	TH, MM	20.4.2
EM, MM	19.6	EM, MM	19.7	EM, MM	19.8
EM, MM	19.9	EM, MM	23.5	EM, MM	23.5.1
EM, MM	23.6.6	CM, MM	21.2.4	CM, MM	21.3
CM, MM	21.4.3	QM, MM	22.2.2	QM, MM	22.2.2
QM, MM	22.2.3	EM, MM	22.7.2	EM, MM	22.10
FD, MM	24.3.2	FD, MM	24.5.3	FD, MM	24.5.4
FD, MM	25.2.3	FD, MM	25.4.4	FD, MM	25.4.5
QM, MM	26.2.3	QM, MM	26.2.4	QM, MM	26.4.5
QM, MM	26.4.6	MM, NL	13.1	MM, CM	12.1
MM	12.6	MM	12.8.1	MM	7.5
MM	7.5.2.1	MM	7.6	MM	7.8.2
MM	13.7.2				

Problems and exercises in molecular dynamics and biological applications

Subject	Section	Subject	Section	Subject	Section
BI, MM	4.4	BI, NL	14.3	BI, NL	14.4
BI, NL	14.5.2	BI, NL	14.5.3	BI, NL	14.10
BI, NL	14.11.1	BI, NL	14.11.4	BI, NL	14.11.5
SP, BI	16.3	BI, NL	14.1	BI, NL	14.9
MD, QM	18.3	MD, QM	18.4	MD	18.4
MM, SP	18				

1.4 This Book's Language: The Python Ecosystem

The codes in this edition of *Computational Physics* employ the computer language *Python*. Previous editions have had their examples in Java, Fortran and C, and used post-simulation tools for visualization. Although we have experienced no general agreement in the computational science community as to the best language for scientific computing, this has not stopped many of the users of each language from declaring it to be the best. Even so, we hereby declare that we have found Python to be the best language yet for teaching CP. Python is free, robust (not easily broken), portable (program run without modifications on various devices), universal (available for most every computer system), has a clean syntax that lets students learn the language quickly, has dynamic typing and high-level, built-in data types that enable getting programs to work quickly without having to declare data types or arrays, count matching braces, or use separate visualization programs. Because Python is interpreted, students can learn the language by executing and analyzing individual commands within an interactive shell, or by running the entire program in one fell swoop. Furthermore, Python brings to scientific computing the availability of a myriad of free packages supporting numerical algorithms, state-of the art, or simple, visualizations and specialized toolkits that rival those in Matlab and Mathematica/Maple. And did we mention, all of this is free?

There are literally thousands of Python packages available, but not to worry, we use only a few for numerical and visualization purposes. Because it is essential to be able to run and modify the example codes in this book, we suggest that you spend the time necessary to get Python to function properly on your computer (and then leave notes as to what you did). For learning Python, we recommend the online tutorials (Ptut, 2014; Pguide, 2014; Plearn, 2014), the books by Langtangen Langtangen (2008) and Langtangen (2009), and the *Python Essential Reference* (Beazley, 2009). For general numerical methods, a book by Press *et al.* (1994) is the standard, and most fun to read, while the NIST Digital Library of Mathematical Functions (NIST, 2014) is probably the most convenient.

Python has developed rapidly since its first implementation in December 1989 (History, 2009). Python's combination of language plus packages is now the standard for the explorative and interactive computing that typifies the present-day scientific research. These rapid developments of Python have also led to a succession of new versions, and the inevitable incompatibilities. Most of the codes in this book were written using Python 2, which was released in 2000, and specifically Python 2.6 with the Visual package (also known as "VPython"). However, there have been major changes to the Python development process as well as in features, and this has led to the release of Python 3.0 in December 2008. Unfortunately, some of the changes in Python 3 were not backward compatible with Python 2.6 and 2.7, and so advances in both Python 2 and 3 and their associated packages have been occurring in parallel. (For our codes, the major difference is in the print statement using a parenthesis in 3, which is not hard to correct.) Furthermore, there have been new versions of operating systems and processors from 32- to 64-bit CPUs, and this also has led to the variety of Python versions and associated packages.

To be honest, we have sometimes felt frustrated by these changes and resulting incompatibilities; however, we are intent on not sharing that! While we will describe the packages and distribution briefly, we indicate here that we have adapted to the real world by having both independent Python 2 and 3 implementations exist on our computers. Specifically, our Visual package programs use Python 3.2, while the others use the *Enthought Canopy Distribution Version 1.3.0*, which at present uses *Python 2.7.3*. (The Visual package is not available in Enthought.)

1.4.1
Python Packages (Libraries)

The Python language plus its family of packages comprise a veritable ecosystem for computing. A package or module is a collection of related methods or classes of methods that are assembled together into a subroutine library.[1] Inclusion of the appropriate packages extends the language to meet the specialized needs of various science and engineering disciplines, and lets one obtain state-of-the-art computing for free. In fact, the May/June 2007 and March/April 2011 issues of *Computing in Science and Engineering* (Perez *et al.*, 2010) focus on scientific computing with Python, and we recommend them.

To use a package named `PackageName`, you include in your Python program either an `import PackageName` or a `from PackageName` statement at the beginning of your program. The `import` statement loads the entire package, which is efficient, but may require you to include the package name as a prefix to the method you want. For example,

```
>>> from visual.graph import *        # Import from visual package
>>> y1 = visual.graph.gcurve(color = blue, delta = 3)      # Use of graph
```

1) The Python Package Index (PYPI, 2014), a repository of free Python packages, currently contains more than 40 000 packages!

Here >>> represents the prompt for a Python shell. Some of the typing can be avoided by assigning a symbol to the package name:

```
>>> import visual.graph as p
>>> y1 = p.gcurve(color = blue, delta = 3)
```

There is also a starred version of `from` that copies *all* of the methods of a package (here Matplotlib called pylab) so that you can leave off prefixes:

```
>>> from pylab import *       # Import all pylab methods
>>> plot(x, y, '-', lw=2)     # A pylab method without prefix
```

1.4.2 This Book's Packages

We are about to describe some of the packages that make Python such a rich environment. If you are anxious to get started now, or worry about getting overwhelmed by the Python packages, you may just want to load *VPython* now and move on to the next chapter. You will need some more stuff to do visualizations and matrices, but you can always upgrade your knowledge when you feel more comfortable with Python.

Because all too often you do not know what you do not know, or what you need to know, we list here a few, basic Python packages and what each does. The packages used in the text are underlined and described more fully later.

Boost.Python A C++ library that enables seamless interoperability between C++ and Python, thus extending the lifetime of legacy codes and making use of the speed of C, www.boost.org/doc/libs/1_55_0/libs/python/doc/.

Cython: C Extensions for Python A superset of the Python language that supports calling C functions and intermixing Python and C for legacy purposes and for high performance, http://cython.org/.

f2py: Fortran to Python Interface Generator that provides connection between Python and Fortran languages; great for steering legacy codes, http://cens.ioc.ee/projects/f2py2e/.

IPython: Interactive Python An advanced *shell* (command line interpreter) that extends Python's basic interpreter IDLE. IPython has enhanced interactivity and interactive visualization capabilities that encourage exploratory computing. IPython also has a browser-based notebook like Mathematica that permits embedded code executions, as well as capabilities for parallel computing, http://ipython.org/.

Matplotlib: Mathematics Plotting Library A 2D and 3D graphics library that uses NumPy (Numerical Python), and produces publication quality figures in a variety of hard copy formats, and permits interactive graphics. Similar to MATLAB's plotting (except Matplotlib is free and doesn't need its license renewed yearly). See Section 1.5.3 for examples and discussion, http://matplotlib.sf.net.

Mayavi Interactive and simplified 3D visualization. Also contains TVTK, a wrapper for the more basic Visualization Tool Kit VTK. ("Mayavi" is Sanskrit for magician.) See Section 1.5.6 for examples and discussion, http://mayavi.sf.net.

Mpmath: Multiprecision Floating Point Arithmetic A pure-Python library for multiprecision floating-point arithmetic for transcendental functions, unlimited exponent sizes, complex numbers, interval arithmetic, numerical integration and differentiation, root-finding, linear algebra, and more, https://code.google.com/p/mpmath/.

NumPy: Numerical Python Permits the use of fast, high-level multidimensional arrays in Python, which are used as the basis for many of the numerical procedures in Python libraries (NumPy, 2013; SciPy, 2014) – the successor to both *Numeric* and *Numarray*. Used by Visual and Matplotlib. *SciPy* extends NumPy. See Sections 6.5, 6.5.1, and 11.2 for examples of NumPy array use.

Pandas: Python Data Analysis Library A collection of high-performance, user-friendly data structures and data analysis tools, http://pandas.pydata.org/.

PIL: Python Imaging Library Image processing and graphics routines for various file formats, www.pythonware.com/products/pil/.

Python The Python standard library, http://python.org.

PyVISA Wrappers for the VISA library providing controls for measurement equipment through various busses from within Python programs, http://pyvisa.readthedocs.org/en/latest/.

SciKits: SciPy Toolkits A collection of toolkits that extend SciPy to special disciplines such as audio processing, financial computation, geosciences, time series analysis, computer vision, engineering, machine learning, medical computing, and bioinformatics, https://scikits.appspot.com/.

SciPy: Scientific Python A basic library for mathematics, science, and engineering. (See SciKits for further extensions.) Provides user-friendly and efficient numerical routines for linear algebra, optimization, integration, special functions, signal and image processing, statistics, genetic algorithms, ODE solutions, and others. Uses NumPy's N-dimensional arrays but also extends NumPy. SciPy essentially provides wrapper for many existing libraries in other languages, such as LAPACK (Anderson *et al.*, 2013) and FFT. The SciPy distribution usually includes Python, NumPy, and f2py, http://scipy.org.

Sphinx Python documentation generator for output in various formats, http://sphinx-doc.org/.

SWIG An interface compiler that connects programs written in C and C++ with scripting languages such as Perl, Python, Ruby, and TCL. Useful for extending the lifetime of legacy codes or for making use of the speed of C, http://swig.org.

SyFi: Symbolic Finite Elements Built on top of the symbolic math library GiNaC, SyFi is used in the finite element solution of PDEs. It provides polygonal domains, polynomial spaces and degrees of freedom as symbolic expressions that are easily manipulated, https://pypi.python.org/pypi/SyFi/.

SymPy: Symbolic Python A system for symbolic mathematics using pure Python (no external libraries) to provide a simple computer algebra system that also includes calculus, differential equations, etc. Similar to Maple or Mathematica, with the Sage package being even more complete. Examples in Section 1.7. See also mpmath, http://sympy.org/.

VisIt Distributed, parallel, visualization tool for visualizing data defined on 2D and 3D structured and unstructured meshes, https://wci.llnl.gov/codes/visit/.

Visual (VPython) Python programming language plus the *visual* 3D graphics module, with the *VIDLE* interactive shell replacing Python's standard *IDLE*. Particularly helpful, even for novices, in creating 3D demonstrations and animations for education. We often use Visual for 2D plots of numerical data and animations. Can be installed separately from Canopy, http://vpython.org/.

1.4.3 The Easy Way: Python Distributions (Package Collections)

Although most Python packages are free, there is a true value for both users and vendors to distribute a collection of packages that have been engineered and tuned to work well together, and that can be installed in one fell swoop. (This is similar to what Red Hat and Debian do with Linux.) These distributions can be thought of as complete Python ecosystems assembled for specific purposes, and are highly recommended. Here we mention four with which we are familiar:

Anaconda A free Python distribution including more than 125 packages for science, mathematics, engineering, and data analysis, including Python, NumPy, SciPy, Pandas, IPython, Matplotlib, Numba, Blaze, and Bokeh. Anaconda is self-described as enterprise-ready for large-scale data processing, predictive analytics, and scientific computing, and permits easy switching between Python 2.6, 2.7, and 3.3. As also true for Canopy, Anaconda installs in its own directory and so runs independently from other Python installations on your computer, https://store.continuum.io/cshop/anaconda/.

Enthought Canopy A comprehensive and complete Python analysis environment with easy installation and updates. The commercial distribution includes more than 150 packages, yet is available for free to academic users. In any case, there is an Express version containing more than 50 packages that is free to everyone. The packages include the IPython, NumPy, SciPy, Matplotlib, Mayavi, scikit, SymPy, Chaco, Envisage, and Pandas, /https://www.enthought.com/products/canopy/.

Python XY A free scientific and engineering development collection of packages for numerical computations, data analysis, and data visualization employing the Qt graphical libraries for GUI development and the Spyder interactive scientific development environment, https://code.google.com/p/pythonxy/.

Sage An amazingly complete collection of open-source packages for mathematical computations, both numerically and symbolically using the IPython interface and notebooks. Sage's stated mission is to create a viable, free, open-source alternative to Magma, Maple, Mathematica, and Matlab, www.sagemath.org/.

1.5 Python's Visualization Tools

If I can't picture it, I can't understand it.

Albert Einstein

In the sections to follow we discuss tools to visualize data produced by simulations and measurements. Whereas other books may choose to relegate this discussion to an appendix, or not to include it at all, we believe that visualization is such an integral part of CP, and so useful for your work in the rest of this book, that we have placed it here, right up front. We describe the use of Matplotlib, Visual (VPython), and Mayavi. VPython makes easy 2D plot, solid geometric figures, and animations. Matplotlib makes very nice 3D (surface) plots, while Mayavi can create state-of-the-art visualizations.

Generalities One of the most rewarding aspects of computing is visualizing the results of calculations. While in the past this was performed with 2D plots, in modern times it is a regular practice to use 3D (surface) plots, volume rendering (dicing and slicing), animations, and virtual reality (gaming) tools. These types of visualizations are often breathtakingly beautiful and may provide deep insights into problems by letting us see and "handle" the functions with which we are working. Visualization also assists in the debugging process, the development of physical and mathematical intuition, and the all-around enjoyment of work.

In thinking about ways to view your results, keep in mind that the point of visualization is to make the science clearer and to communicate your work to others. Then it follows that you should make all figures as clear, informative, and self-explanatory as possible, especially if you will be using them in presentations without captions. This means labels for curves and data points, a title, and labels on the axes.[2] After this, you should look at your visualization and ask whether there are better choices of units, ranges of axes, colors, style, and so on, that might get the message across better and provide better insight. And try to remember that those colors which look great on your monitor may turn into uninformative grays when printed. Considering the complexity of human perception and cognition,

2) Although this may not need saying, place the independent variable x along the abscissa (horizontal), and the dependent variable $y = f(x)$ along the ordinate.

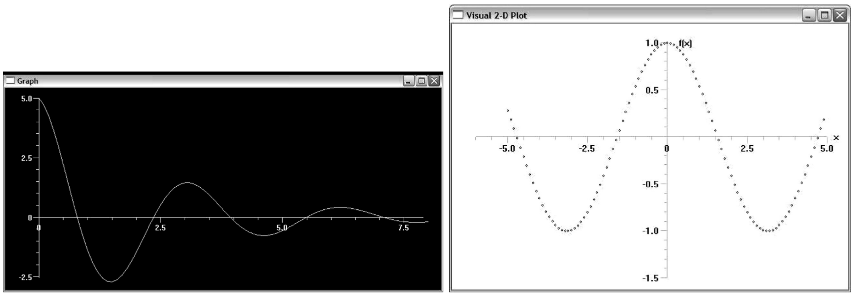
there may not be a single best way to visualize a particular data set, and so some trial and error may be necessary to “see” what works best.

Listing 1.1 EasyVisual.py produces two different 2D plot using the Visual package.

```
# EasyVisual.py:        Simple graph object using Visual

from visual.graph import *                    # Import Visual

Plot1 = gcurve(color = color.white)           # gcurve method

for x in arange(0., 8.1, 0.1):                # x range
    Plot1.plot( pos = (x, 5.*cos(2.*x)*exp(-0.4*x)) ) # Plot pts

graph1 =  gdisplay(width=600, height=450,\
    title='Visual 2D Plot', xtitle='x', ytitle='f(x)',\
     foreground = color.black, background = color.white)

Plot2 = gdots(color = color.black)            # Dots

for x in arange( -5., +5, 0.1 ):
    Plot2.plot(pos = (x, cos(x)))
```

1.5.1 Visual (VPython)’s 2D Plots

As indicated in the description of packets, VPython (Python plus the Visual package) is a simple way to get started with Python and visualizations.[3] The Visual package is useful for creating 3D solids, 2D plots, and animations. For example, in Figure 1.2, we present two plots produced by the program `EasyVisual.py` in Listing 1.1. Notice that the plotting technique is to create first a plot object, and then to add the points to the object, one by one. (In contrast, Matplotlib creates a vector of points and then plots the entire vector.)



(a) (b)

Figure 1.2 Screen dumps of two *x*–*y* plots produced by `EasyVisual.py` using the Visual package. Plot (a) uses default parameters, while plot (b) uses user-supplied options.

3) Because Visual is not one of the Canopy packages, to run our Visual programs you would need to install the Visual package and the version of Python that runs with it, even if you have Canopy installed. There is no problem doing this because VPython and Canopy go into different folders/directories.

EasyVisual.py is seen to create two plot objects, Plot1 and Plot2, with the plot method used to plot each object. Plot1 uses the gcurve method with no options specified other than the color of the curve (white). We obtain (Figure 1.2a) a connected curve by default, but no labels. In contrast, Plot2 uses the gdisplay method to set the display characteristics for the plot to follow, and then gdots to draw the data as points (Figure 1.2b). You use a gdisplay plotting object before you create a gcurve graph object, in order to set the size, position, and title for the graph window, to specify titles for the x and y axes, and to specify maximum values for each axis. The gdisplay arguments are self-explanatory, with the width and height given in pixels. Because we set Plot2= gdots(), only points are plotted.

Listing 1.2 3GraphVisual.py produces a 2D x–y plot with the Matplotlib and NumPy packages.

```
# 3GraphVisual.py: 3 plots in the same figure, with bars, dots and curve

from visual import *
from visual.graph import*

string = "blue: sin^2(x), white: cos^2(x), red: sin(x)*cos(x)"
graph1 = gdisplay(title=string, xtitle='x', ytitle='y')

y1 = gcurve(color=color.yellow, delta=3)          # curve
y2 = gvbars(color=color.white)                    # vertical bars
y3 = gdots(color=color.red, delta=3)              # dots

for x in arange(-5, 5, 0.1):                       # arange for floats
    y1.plot( pos=(x, sin(x)*sin(x)) )
    y2.plot( pos=(x, cos(x)*cos(x)/3.) )
    y3.plot( pos=(x, sin(x)*cos(x)) )
```

Note that the Python codes are listed within shaded boxes with some formatting to improve readability. For example, see Listing 1.2. Note that we have structured the codes so that a line is skipped before major elements like functions, and that indentations indicate structures in Python (where Java and C may use braces).

It is often a good idea to place several plots in the same figure. The program 3GraphVisual.py in Listing 1.2 does that and produces the graph in Figure 1.3a. There are white vertical bars created with gvbars, red dots created with gdots, and a yellow curve created with gcurve (colors appear only as shades of gray in the paper text). Also note in 3GraphVisual.py that we avoid having to include the package name as a prefix to the commands by starting the program with import visual.graph as vg. This both imports Visual's graphing package and assigns the symbol vg to visual.graph.

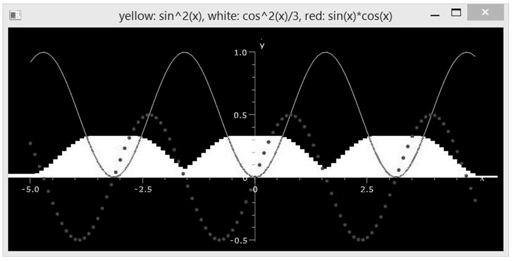
1.5.1.1 VPython's 3D Objects

Listing 1.3 3Dshapes.py produces a sample of VPython's 3D shapes.

```
# 3Dshapes.py: Some 3D Shapes of VPython

from visual import *

graph1 = display(width=500, height=500, title='VPython 3D Shapes',
     range=10)
```

```
sphere(pos=(0,0,0), radius=1, color=color.green)
sphere(pos= (0,1,-3), radius=1.5, color=color.red)
arrow(pos=(3,2,2), axis=(3,1,1), color=color.cyan)
cylinder(pos=(-3,-2,3), axis=(6,-1,5), color=color.yellow)
cone(pos=(-6,-6,0), axis=(-2,1,-0.5), radius=2, color=color.magenta)
helix(pos=(-5,5,-2), axis=(5,0,0), radius=2, thickness=0.4,
     color=color.orange)
ring(pos=(-6,1,0), axis=(1,1,1), radius=2, thickness=0.3,
     color=(0.3,0.4,0.6))
box(pos=(5,-2,2), length=5, width=5, height=0.4, color=(0.4,0.8,0.2))
pyramid(pos=(2,5,2), size=(4,3,2), color=(0.7,0.7,0.2))
ellipsoid(pos=(-1,-7,1), axis=(2,1,3), length=4, height=2, width=5,
     color=(0.1,0.9,0.8))
```

One way to make simulations appear more realistic is to use 3D solid shapes, for example, a sphere for a bouncing ball rather than just a dot. VPython can produce a variety of 3D shapes with one-line commands, as shown in Figure 1.4, and as produced by the code in Listing 1.3. To make the ball bounce, you would need to vary the `position` variable according to some kinematic equations.



(a)

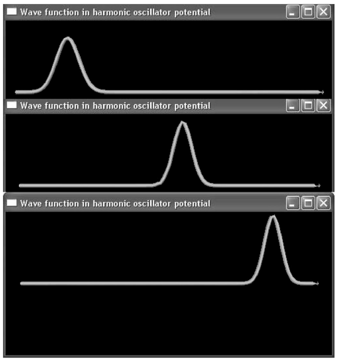

Wave function in harmonic oscillator potential
Wave function in harmonic oscillator potential
Wave function in harmonic oscillator potential
(b)

Figure 1.3 (a) Output from the program 3GraphVisual.py that places three different types of 2D plots on one graph using Visual. (b) Three frames from a Visual animation of a quantum mechanical wave packet produced with HarmosAnimate.py.

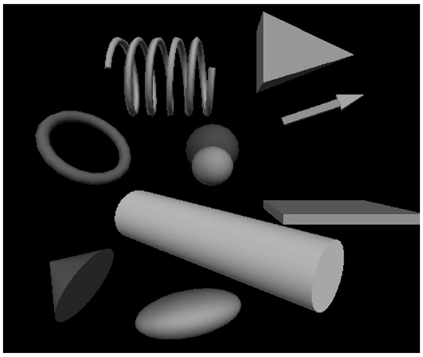


Figure 1.4 Some 3D shapes created with single commands in VPython.

1.5.2 VPython's Animations

Creating animations with Visual is essentially just making the same 2D plot over and over again, with each one at a slightly differing time, and then placing the plots on top of each other. When performed properly, this gives the impression of motion. Several of our sample codes produce animations, for example, `HarmosAnimate.py` and `3Danimate.py`. Three frames produced by `HarmosAnimate.py` are shown in Figure 1.3b. The major portions of these codes deal with the solution of PDEs, which need not concern us yet. The part which makes the animation is simple:

```
PlotObj= curve(x=xs, color=color.yellow, radius=0.1)
...
while True:                                          # Runs forever
   rate(500)
   psr[1:-1] = ...
   psi[1:-1] = ..
   PlotObj.y = 4*(psr**2 + psi**2)
```

Here `PlotObj` is a curve that continually gets built from within a while loop and thus appears to be moving. Note that being able to plot points individually without having to store them all in an array for all times keeps the memory demand of the program quite small and leads to fast programs.

Listing 1.4 EasyMatPlot.py produces a, 2D *x*–*y* plot using the Matplotlib package (which includes the NumPy package).

```
# EasyMatPlot.py: Simple use of matplotlib's plot command

from pylab import *                                   # Load Matplotlib

Xmin = -5.;    Xmax = +5.;    Npoints= 500
DelX = (Xmax - Xmin) / Npoints
x = arange(Xmin, Xmax, DelX)
y =  sin(x) * sin(x*x)                          # function of x array

print ('arange => x[0], x[1],x[499]=%8.2f %8.2f %8.2f'
     %(x[0],x[1],x[499]))
print ('arange => y[0], y[1],y[499]=%8.2f %8.2f %8.2f'
     %(y[0],y[1],y[499]))
print ("\n Now doing the plotting thing, look for Figure 1 on desktop" )
xlabel('x');      ylabel('f(x)');      title(' f(x) vs x')
text(-1.75, 0.75, 'MatPlotLib \n Example')            # Text on plot
plot(x, y, '-', lw=2)
grid(True)                                               # Form grid
show()
```

1.5.3 Matplotlib's 2D Plots

Matplotlib is a powerful plotting package that lets you make 2D and 3D graphs, histograms, power spectra, bar charts, error charts, scatter plots, and more, all directly from within your Python program. Matplotlib is free, uses the sophisticated numerics of NumPy and LAPACK (Anderson *et al.*, 2013), and, believe it or not, is easy to use. Specifically, Matplotlib uses the NumPy `array` (vector) object to store the data to be plotted. In Chapter 6, we talk at more length about NumPy arrays, so you may want to go there soon to understand arrays better.

Matplotlib commands are by design similar to the plotting commands of MATLAB, a commercial problem-solving environment that is particularly popular in engineering. As is true for MATLAB, Matplotlib assumes that you have placed the x and y values that you wish to plot into 1D arrays (vectors), and then plots these vectors in one fell swoop. This is in contrast to Visual, which first creates a plot object and then adds points to the object one by one. Because Matplotlib is not part of standard Python, you must import the entire Matplotlib package, or the individual methods you use, into your program. For example, on line 2 of `EasyMatPlot.py` in Listing 1.4 (line numbers are in the dark shading on the right), we import Matplotlib as the `pylab` library:

```
from pylab import *                        # Load Matplotlib
```

Then, on lines 6 and 7 we calculate and input arrays of the x and y values

```
x = arange(Xmin, Xmax, DelX)          # Form x array in range with increment
y =  -sin(x)*cos(x)                   # Form y array as function of x array
```

As you can see, NumPy's `arange` method constructs an array covering "a range" between `Xmax` and `Xmin` in steps of `DelX`. Because the limits are floating-point numbers, so also will be the x_i's. And because `x` is an array, `y = -sin(x)*cos(x)` is automatically one too! The actual plotting is performed on line 14 with a dash "–" used to indicate a line, and `lw` = 2 to set its width. The result is shown in Figure 1.5a with the desired labels and title. The `show()` command produces the graph on your desktop. More commands are given in Table 1.3. We suggest you try out some of the options and types of plots possible.

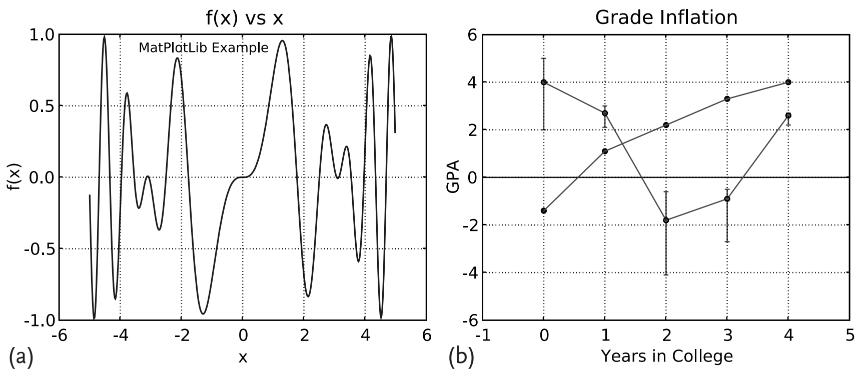


Figure 1.5 Matplotlib plots. (a) Output of EasyMatPlot.py showing a simple, x–y plot. (b) Output from GradesMatPlot.py that places two sets of data points, two curves, and unequal upper and lower error bars, all on one plot.

Table 1.3 Some common Matplotlib commands.

Command	Effect	Command	Effect
plot(x, y, '-', lw=2)	$x-y$ curve, line width 2	myPlot.setYRange(-8., 8.)	Set y range
show()	Show output graph	myPlot.setSize(500, 400)	Size in pixels
xlabel('x')	x-axis label	pyplot.semilogx	Semilog x plot
ylabel('f(x)')	y-axis label	pyplot.semilogy	Semilog y plot
title('f vs x')	Add title	grid(True)	Draw grid
text(x, y, 's')	Add text s at (x, y)	myPlot.setColor(false)	Black and White
myPlot.addPoint (0,x,y,true)	Add (x, y) to set 0 connect	myPlot.setButtons(true)	For zoom button
myPlot.addPoint (1,x,y, false)	Add (x, y) to 1, no connect	myPlot.fillPlot()	Fir ranges to data
pyplot.errorbar	Point + error bar	myPlot.setImpulses(true,0)	Vertical lines, set 0
pyplot.clf()	Clear current figure	pyplot.contour	Contour lines
pyplot.scatter	Scatter plot	pyplot.bar	Bar charts
pyplot.polar	Polar plot	pyplot.gca	For current axis
myPlot.setXRange(-1., 1.)	Set x range	pyplot.acorr	Autocorrelation

Listing 1.5 GradesMatPlot.py produces a, 2D x–y plot using the Matplotlib package.

```
# Grade.py: Using Matplotlib's plot command with multi data sets & curves

import pylab as p                                              # Matplotlib
from numpy import*

p.title('Grade Inflation')                              # Title and labels
p.xlabel('Years in College')
p.ylabel('GPA')

xa = array([-1, 5])                                   # For horizontal line
ya = array([0, 0])                                        #  "        "
p.plot(xa, ya)                                      # Draw horizontal line

x0 = array([0, 1, 2, 3, 4])                             # Data set 0 points
y0 = array([-1.4, +1.1, 2.2, 3.3, 4.0])
p.plot(x0, y0, 'bo')                            # Data set 0 = blue circles
p.plot(x0, y0, 'g')                                     # Data set 0 = line

x1 = arange(0, 5, 1)                                    # Data set 1 points
y1 = array([4.0, 2.7, -1.8, -0.9, 2.6])
p.plot(x1, y1, 'r')

errTop = array([1.0, 0.3, 1.2, 0.4, 0.1])        # Asymmetric error bars
errBot = array([2.0, 0.6, 2.3, 1.8, 0.4])
p.errorbar(x1, y1, [errBot, errTop], fmt = 'o')         # Plot error bars

p.grid(True)                                                   # Grid line
p.show()                                           # Create plot on screen
```

In Listing 1.5, we give the code `GradesMatplot.py`, and in Figure 1.5b we show its output. This is not a simple plot. Here we repeat the `plot` command several times in order to plot several data sets on the same graph, and to plot both the data points and the lines connecting them. On line 3, we import Matplotlib (pylab), and on line 4 we import NumPy, which we need for the `array` command. Because we have imported two packages, we add the `pylab` prefix to the `plot` commands so that Python knows which package to use.

In order to place a horizontal line along $y = 0$, on lines 10 and 11 we create a data set as an array of x values, $-1 \le x \le 5$, and a corresponding array of y values, $y_i \equiv 0$. We then plot the horizontal on line 12. Next we place four more curves in the figure. First on lines 14–15, we create data set 0, and then plot the points as blue circles (`'bo'`), and connect the points with green (`'g'`) lines (the color will be visible on a computer screen, but will appear only as shades of gray in print). On lines 19–21, we create and plot another data set as a red (`'r'`) line. Finally, on lines 23–25, we define unequal lower and upper error bars and place them on the plot. We finish by adding grid lines (line 27) and *showing* the plot on the screen.

Listing 1.6 MatPlot2figs.py produces the two figures shown in Figure 1.6. Each figure contains two plots with one Matplotlib figure.

```python
# MatPlot2figs.py: plot of 2 subplots on 1 fig & 2 separate figs

from pylab import *                                          # Load matplotlib

Xmin = -5.0;      Xmax =  5.0;       Npoints= 500
DelX= (Xmax-Xmin)/Npoints                                          # Delta x
x1 = arange(Xmin, Xmax, DelX)                                      # x1 range
x2 = arange(Xmin, Xmax, DelX/20)                         # Different x2 range
y1 =  -sin(x1)*cos(x1*x1)                                        # Function 1
y2 =   exp(-x2/4.)*sin(x2)                                       # Function 2
print("\n Now plotting, look for Figures 1 & 2 on desktop")
#        Figure 1
figure(1)
subplot(2,1,1)                                      # 1st subplot in first figure
plot(x1, y1, 'r', lw=2)
xlabel('x');      ylabel( 'f(x)' );      title( '-sin(x)*cos(x^2)' )
grid(True)                                                        # Form grid
subplot(2,1,2)                                      # 2nd subplot in first figure
plot(x2, y2, '-', lw=2)
xlabel('x')                                                     # Axes labels
ylabel( 'f(x)' )
title( 'exp(-x/4)*sin(x)' )

#        Figure 2
figure(2)
subplot(2,1,1)                                        # 1st subplot in 2nd figure
plot(x1, y1*y1, 'r', lw=2)
xlabel('x');       ylabel( 'f(x)' );    title( 'sin^2(x)*cos^2(x^2)' )

    # form grid
subplot(2,1,2)                                        # 2nd subplot in 2nd figure
plot(x2, y2*y2, '-', lw=2)
xlabel('x');      ylabel( 'f(x)' );      title( 'exp(-x/2)*sin^2(x)' )
grid(True)

show()                                                          # Show graphs
```

Often the science is clearer if there are several curves in one plot, and, several plots in one figures. Matplotlib lets you do this with the `plot` and the `subplot` commands. For example, in `MatPlot2figs.py` in Listing 1.6 and Figure 1.6, we have

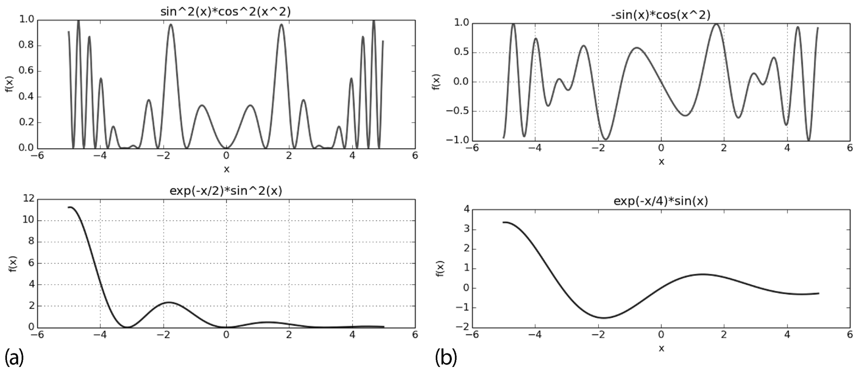


Figure 1.6 (a,b) Columns show two separate outputs, each of two figures, produced by MatPlot2figs.py. (We used the slider button to add some space between the plots.)

placed two curves in one plot, and then output two different figures, each containing two plots. The key here is the repetition of the `subplot` command:

```
figure(1)              # The 1st figure
subplot(2,1,1)         # 2 rows, 1 column, 1st subplot
subplot(2,1,2)         # 2 rows, 1 column, 2nd subplot
```

The listing is self-explanatory, with sections that set the plotting limits, that creates each figure, and then creates the grid.

Listing 1.7 PondMatPlot.py produces the scatter plot and the curve shown in Figure 5.5 in Chapter 5.

```
#  PondMatPlot.py: Monte-Carlo integration via vonNeumann rejection

import numpy as np
import matplotlib.pyplot as plt

N = 100
x1 = np.arange(0, 2*np.pi+2*np.pi/N,2*np.pi/N)
fig,ax = plt.subplots()
y1 = x1 * np.sin(x1)**2                         # Integrand
ax.plot(x1, y1, 'c', linewidth=4)
ax.set_xlim ((0, 2*np.pi))
ax.set_ylim((0, 5))
ax.set_xticks([0, np.pi, 2*np.pi])
ax.set_xticklabels(['0', '$\uppi$','2$\uppi$'])
ax.set_ylabel('$f(x) = x\,\sin^2 x$', fontsize=20)
ax.set_xlabel ('x',fontsize=20)
fig.patch.set_visible(False)
xi=[];  yi=[];  xo=[];  yo=[]

def fx (x):                                    # Integrand
    return x*np.sin(x)**2

j = 0                                # Inside curve counter
Npts = 3000
analyt = np.pi**2
xx = 2.* np.pi * np.random.rand(Npts)     # 0 =< x <= 2pi
yy = 5*np.random.rand(Npts)               # 0 =< y <= 5
for i in range(1,Npts):
        if (yy[i] <= fx(xx[i])):                    # Below curve
            if (i <=100): xi.append(xx[i])
            if (i <=100): yi.append(yy[i])
            j +=1
        else:
            if (i <=100): yo.append(yy[i])
            if (i <=100): xo.append(xx[i])

    boxarea = 2. * np.pi *5.                    # Box area
    area = boxarea*j/(Npts-1)                   # Area under curve
    ax.plot(xo,yo,'bo',markersize=3)
    ax.plot(xi,yi,'ro',markersize=3)
    ax.set_title('Answers:  Analytic = %5.3f, MC = %5.3f'%(analyt,area))
plt.show()
```

Scatter Plots Sometimes we need a scatter plot of data points, and maybe even a curve thrown in as well. In Figure 5.5 in Chapter 5, we show such a plot created with the code `PondMapPlot.py` in Listing 1.7. The key statements here are of the form `ax.plot(xo, yo, 'bo', markersize=3)`, which in this case adds a blue point (on screen) of size 3.

1.5.4 Matplotlib's 3D Surface Plots

A 2D plot of the potential $V(r) = 1/r$ vs. r is fine for visualizing the radial dependence of the potential field surrounding a single charge, but if you want to visualize a dipole potential such as $V(x, y) = (B + C(x^2 + y^2)^{-3/2})x$, you need a 3D visualization. We get that by creating a world in which the z dimension (mountain height) is the value of the potential, and the x and y axes define the plane below the mountain. Because the surface we are creating is a 3D object, it is not truly possible to draw it on a flat screen, and so different techniques are used to give the impression of three dimensions to our brains. We do that by rotating the object (by grabbing it with your mouse), shading it, employing parallax, and other tricks.

Listing 1.8 Simple3Dplot.py produces the Matplotlib 3D surface plots in Figure 1.7.

```
# Simple3Dplot.py: matplotlib 3D plot you can rotate and scale via mouse

import matplotlib.pylab  as p
from mpl_toolkits.mplot3d import Axes3D

print "Please be patient, I have packages to import & points to plot"
delta = 0.1
x = p.arange( -3., 3., delta )
y = p.arange( -3., 3., delta )
X, Y = p.meshgrid(x, y)
Z = p.sin(X) * p.cos(Y)                               # Surface height

fig = p.figure()                                      # Create figure
ax = Axes3D(fig)                                      # Plots axes
ax.plot_surface(X, Y, Z)                              # Surface
ax.plot_wireframe(X, Y, Z, color = 'r')               # Add wireframe
ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('Z')

p.show()                                              # Output figure
```

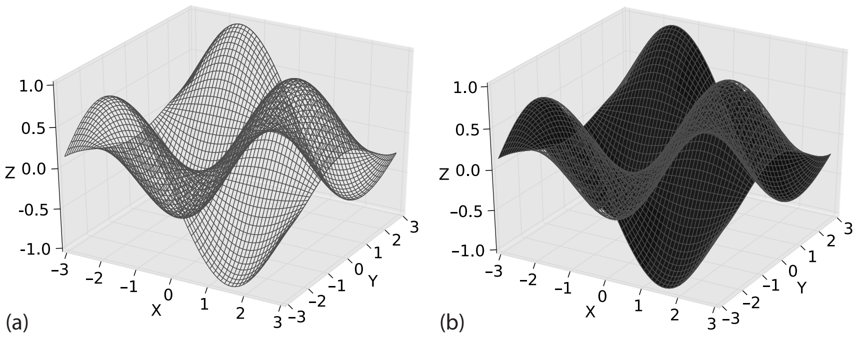


Figure 1.7 (a) A 3D wire frame. (b) A surface plot with wire frame. Both are produced by the program Simple3dplot.py using Matplotlib.

In Figure 1.7a, we show a wire-frame plot and in Figure 1.7b, a surface-plus-wire-frame plot. These are obtained from the program `Simple3Dplot.py` in Listing 1.8. Note that there is an extra import of `Axes3D` from the Matplotlib tool kit needed for 3D plotting. Lines 8 and 9 are the usual creation of x and y arrays of floats using `arange`. Line 11 uses the `meshgrid` method to set up the entire coordinate matrix grid from the x and y coordinate vectors with a vector call, and line 12 constructs the entire Z surface with another vector operation. The remaining of the program is self-explanatory, with `fig` being the plot object, `ax` the 3D axes object, and `plot_wireframe` and `plot_surface` creating wire frame and surface plots, respectively.

Another type of 3D plot is particularly useful when examining data of the form (x_i, y_j, z_k), is a scatter plot into the 3D (x, y, z) volume. In Listing 1.9, we give the program `Scatter3dPlot.py` that created the plot in Figure 1.8. This program, which is taken from the Matplotlib documentation, uses the NumPy random number generator, with the 111 notation being a hand-me-down from MATLAB indicating a $1 \times 1 \times 1$ grid.

Listing 1.9 Scatter3dPlot.py produces a 3D scatter plot using Matplotlib 3D tools.

```python
" Scatter3dPlot.py    from matplotlib examples"

import numpy as np
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt

def randrange(n, vmin, vmax):
    return (vmax-vmin)*np.random.rand(n) + vmin

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
n = 100
for c, m, zl, zh in [('r', 'o', -50, -25), ('b', '^', -30, -5)]:
    xs = randrange(n, 23, 32)
    ys = randrange(n, 0, 100)
    zs = randrange(n, zl, zh)
    ax.scatter(xs, ys, zs, c=c, marker=m)
```

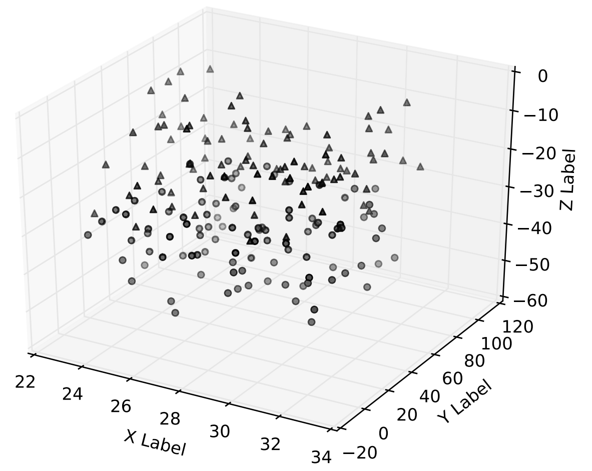


Figure 1.8 A 3D scatter plot produced by the program Scatter3dPlot.py using Matplotlib.

```
ax.set_xlabel('X Label')
ax.set_ylabel('Y Label')
ax.set_zlabel('Z Label')

plt.show()
```

Finally, the program `FourierMatplot.py`, written by Oscar Restrepo, performs a Fourier reconstruction of a saw tooth wave, with the number of waves included controlled by the viewer via a slider bar, as shown in Figure 1.9. (We discuss the mathematics of Fourier transforms in Chapter 12.) The slider method is included via the extra lines:

```
from   matplotlib.widgets import Slider
...
snumwaves = Slider(axnumwaves, '# Waves', 1, 20, valinit=T)
...
snumwaves.on_changed(update)
```

1.5.5 Matplotlib's Animations

Matplotlib also can do animations, although not as simply as VPython. The Matplotlib example page shows a number of them. We include some Matplotlib animation codes in the PythonCodes/Visualizations directory, and show a sample code for the heat equation in Listing 1.10. Here too most of the code deals with solving a PDE, which need not interest us yet. The animation is carried out at the bottom of the code.

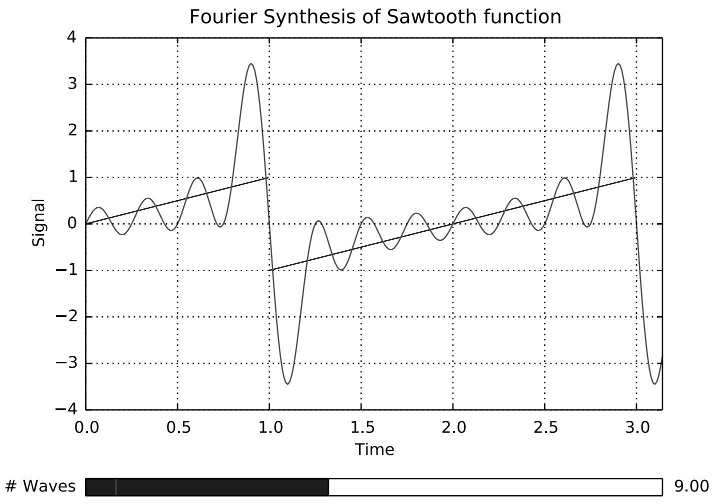


Figure 1.9 A comparison of a saw tooth function to the sum of its Fourier components, with the number of included waves varied interactively by a Matplotlib slider. FourierMatplot.py, which produced this output, was written by Oscar Restrepo.

Listing 1.10 EqHeatAnimateMatPlot.py produces an animation of a cooling bar using Matplotlib.

```
# EqHeat.py Animated heat equation soltn via fine differences

from numpy import *
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation

Nx = 101
Dx = 0.01414
Dt = 0.6
KAPPA = 210.                                        # Thermal
     conductivity
SPH = 900.                                          # Specific heat
RHO = 2700.                                         # Density
cons = KAPPA/(SPH*RHO)*Dt/(Dx*Dx);
T = np.zeros( (Nx, 2), float)                     # Temp @ first 2 times
def init():
   for ix in range (1, Nx - 1):                     # Initial temperature
       T[ix, 0] = 100.0;

   T[0, 0] = 0.0                                    # Bar ends T = 0
   T[0, 1] = 0.
   T[Nx - 1, 0] = 0.
   T[Nx - 1, 1] = 0.0
init()
k=range(0,Nx)
fig=plt.figure()                                      # Figure to plot
# select axis; 111: only one plot, x,y, scales given
ax = fig.add_subplot(111, autoscale_on=False, xlim=(-5, 105), ylim=(-5,
     110.0))
ax.grid()                                                  # Plot
     grid
plt.ylabel("Temperature")
plt.title("Cooling of a bar")
```

```
line , = ax.plot(k, T[k,0],"r", lw=2)
plt.plot([1,99],[0,0],"r",lw=10)
plt.text(45,5,'bar',fontsize=20)

def animate(dum):
   for ix in range (1, Nx - 1):
      T[ix, 1] = T[ix, 0] +  cons*(T[ix + 1, 0] + T[ix - 1, 0] -
           2.0*T[ix, 0])
   line.set_data(k,T[k,1] )
   for ix in range (1, Nx - 1):
        T[ix, 0] = T[ix, 1]                    # Row of 100 positions at
             t = m
   return line,

ani = animation.FuncAnimation(fig, animate,1)          # Animation
plt.show()
```

1.5.6 Mayavi's Visualizations Beyond Plotting*

This section on Mayavi is indicated as optional because we do not use it in our sample programs. However, we recommend that, at least, the reader browse through it in order to obtain some ideas about the next level of Python visualization.

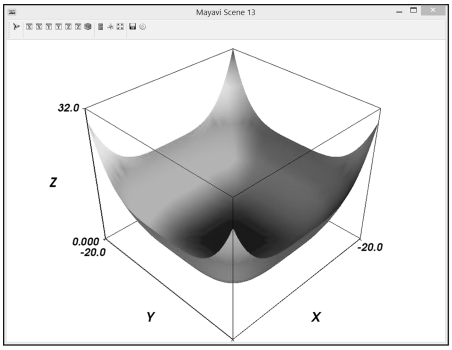
Although Matplotlib is excellent for plotting functions vs. one or two of its variables, it is not designed to do the sculpture-like 3D visualizations of functions of three or more variables that are often displayed by supercomputer centers. Mayavi (Sanskrit for “magician”) is designed for this next level of visualization. Mayavi is open source, tightly integrated with Python and included in the Canopy distribution.

Mayavi consists of two different packages and two different interfaces to those packages. The package we illustrate here is the set of Matlab- or Mathematica-like commands that operate at a fairly high level of abstraction and works naturally with NumPy arrays. The other package is a set of VTK (Visual Tool Kit) primitives that may be more appropriate for developing your own, research-specific, visualization modules. Even with the high-level package, you have the choice of interacting with Mayavi via scripting from within your Python program (what we demonstrate) or via a stand-alone application that runs separately from your programs.

We will now show a few examples derived from the Enthought tutorial. We start by having Mayavi produce a standard surface plot of $z(x, y) = x^4 + y^4$:

```
import numpy; import Matplotlib; import matplotlib.pyplot
import mayavi;import mayavi.mlab

X, Y = numpy.mgrid[-2:2: 0.1, -2:2: 0.1];               Z = X**4 + Y**4

mayavi.mlab.surf(Z);                          mayavi.mlab.axes()
mayavi.mlab.outline();                        mlab.show()
```

You see here that we use NumPy's `numpy.mgrid` method to set up the X and Y arrays, and then set up the Z array with a vectorized evaluation of $X^4 + Y^4$. Then we use Mayavi to create the Z surface, to draw the axes and to outline the surface



(a)

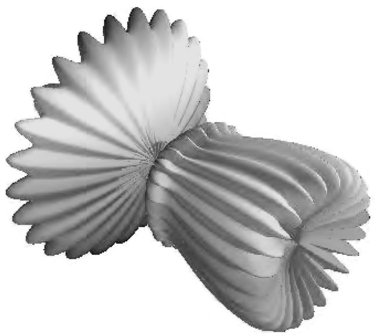

(b)

Figure 1.10 (a) A Mayavi surface plot of the function $z = x^4 + y^4$ as seen in the screen viewer. (b) A rotatable visualization of a spherical harmonic $Y_l^m(\theta, \phi)$ in which the radial distance represents the value of the function.

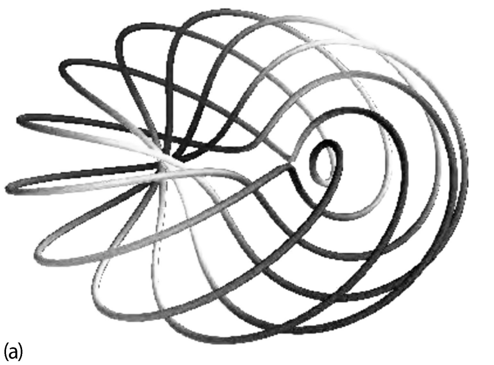
with a box. Finally, there is an important call to `mlab.show()` to show the visualization in a display box such as that in Figure 1.10a. This display box is seen (well, if enlarged) to contain a number of (too small) buttons that lets you produce different views and sizes, insert directional arrows, save the file in various formats to disk, edit properties of the visualization, and open the *pipeline* window. The pipeline window shows, and lets the user control, the various stages of a visualization: loading the data into a data source object, transforming the data with filters, and visualizing it with modules.

Now we go beyond the direct plotting of a function's values to the creation of a visualization of a spherical harmonic function $Y_l^m(\theta, \phi)$ that is defined over the surface of a sphere (Figure 1.10b):

```
from numpy import pi, sin, cos, mgrid
from mayavi import mlab
dphi, dtheta = pi/250.0, pi/250.0
[phi,theta] = mgrid[0:pi+dphi*1.5:dphi,0:2*pi+dtheta*1.5:dtheta]
m0 = 4; m1 = 3; m2 = 2; m3 = 3; m4 = 6; m5 = 2; m6 = 6; m7 = 4;
r = sin(m0*phi)**m1 + cos(m2*phi)**m3 + sin(m4*theta)**m5 +
      cos(m6*theta)**m7
x = r*sin(phi)*cos(theta); y = r*cos(phi)               # Function
z = r*sin(phi)*sin(theta)                               # Projections
 # View data
s = mlab.mesh(x, y, z)
mlab.show()
```

Because we do not have four dimensions to use, we take the values of $Y_l^m(\theta, \phi)$ at various grid points and plot those values as the radial distances from the origin for each value of θ and ϕ. The new element here is the statement `s = mlab.mesh(x, y, z)` that produces a mesh throughout 3D space, and then the projection of the radius into its (x, y, z) components.

In the next example, we start with a data set in the form of (x_i, y_j, z_k) values and connect the points with *tubes* of various colors:


(a)

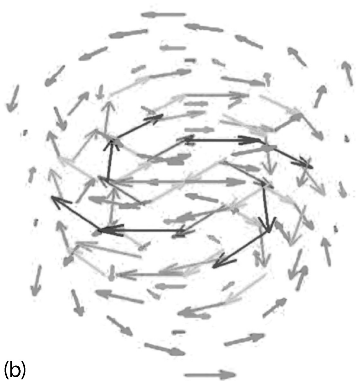

(b)

Figure 1.11 (a) A Mayavi visualization in which tubes are used to connect a set of data points. (b) A Mayavi visualization using arrows (glyphs) to represent a vector field.

```
import numpy; import mayavi
from mayavi.mlab import *

n_mer, n_long = 6, 11 ; pi =numpy.pi
dphi  =  pi / 1000.0
phi = numpy.arange(0.0, 2 * pi + 0.5 * dphi, dphi)
mu = phi * n_mer
x = numpy.cos(mu) * (1 + numpy.cos(n_long * mu / n_mer) * 0.5)
y = numpy.sin(mu) * (1 + numpy.cos(n_long * mu / n_mer) * 0.5)
z = numpy.sin(n_long * mu / n_mer) * 0.5

plot3d(x, y, z, numpy.sin(mu), tube_radius=0.025, colormap='Spectral')
mayavi.mlab.show()
```

The new command here is `plot3d`, which is seen in Figure 1.11b to produce rainbow colored (`'Spectral'`) tubes connecting the data points. The `arange` command sets up the array of `phi` values, and then the arrays of `x`, `y`, `z`, `mu` and `sin(mu)` values all follow.

A popular style of visualization for vector fields is one in which arrows (*glyphs*) are drawn at various points in space with the directions of the arrows indicating the directions of the field, and with the length of the arrows indicating its strengths. Here we create such a visualization and show its output in Figure 1.11b:

```
import numpy
from mayavi.mlab import *

x, y, z = numpy.mgrid[-2:3, -2:3, -2:3]
r = numpy.sqrt(x ** 2 + y ** 2 + z ** 4)
u = y * numpy.sin(r) / (r + 0.001)
v = -x * numpy.sin(r) / (r + 0.001)
w =  4*numpy.zeros_like(z)

quiver3d(x, y, z, u, v, w, line_width=3, scale_factor =1.5)
show()
```

As before, we use NumPy to set an (x, y, z) grid. Then we set up an array of r values as an intermediate function of (x, y, z), and finally set up arrays of the (u, v, w)

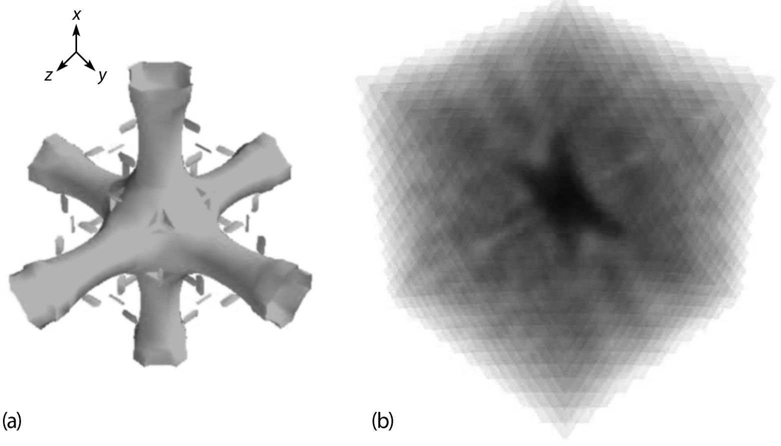


Figure 1.12 (a) A Mayavi visualization of the contours of the scalar field $\phi(x, y, z) = \sin(xyz)/xyz$. (b) A volume rendering of the same scalar field.

components of the vector field as functions of the other arrays. The new command here is `quiver3d` which provides a collection of arrows (cute name).

If the field we wish to visualize is a scalar field, such as $\phi(x, y, z) = \sin(xyz)/xyz$, then the appropriate visualization would be an iso-surface (a 3D contour plot of equal values) throughout a 3D space. We do that with the `contour3d` command:

```
import numpy as np
from mayavi import mlab
x, y, z = np.ogrid[-10:10:20j, -10:10:20j, -10:10:20j]
scalar = np.sin(x*y*z)/(x*y*z)
mlab.contour3d(scalar)
mlab.show()
```

Figure 1.12a shows the output, which is periodic, but not obviously trigonometric.

We now take our visualization of the same scalar field and show how some other Mayavi methods yield different views of the field. First, a volume rendering to produce the nebulous view in Figure 1.12b:

```
import numpy as np
from mayavi import mlab
x, y, z = np.ogrid[-10:10:20j, -10:10:20j, -10:10:20j]
s = np.sin(x*y*z)/(x*y*z)
mlab.pipeline.volume(mlab.pipeline.scalar_field(s))
mlab.show()
```

Next, we take the same field and replace the `mlab.contour3d(s)` command with the pipeline command:

```
mlab.pipeline.volume(mlab.pipeline.scalar_field(s))
```

This produces the nebulous visualization in Figure 1.12b. Next, we produce the visualization in Figure 1.13a by having some planes cut through the scalar field:

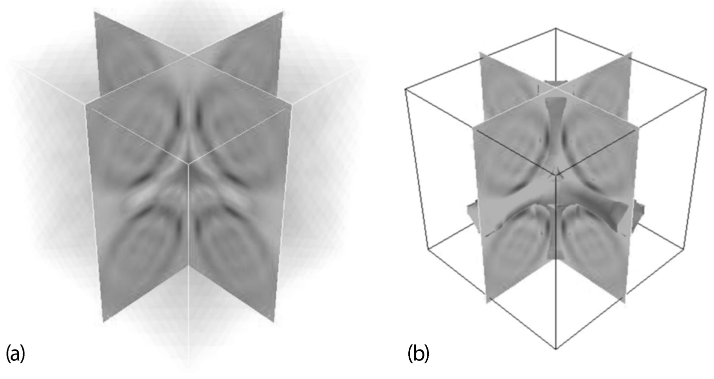


Figure 1.13 (a) A Mayavi visualization of the same scalar field $\phi(x, y, z) = \sin(xyz)/xyz$ using cut planes. (b) A visualization of the same scalar field combining cut planes and contours.

```
import numpy as np
from mayavi import mlab
x, y, z = np.ogrid[-10:10:20j, -10:10:20j, -10:10:20j]
scalar = np.sin(x*y*z)/(x*y*z)
mlab.pipeline.image_plane_widget(mlab.pipeline.scalar_field(scalar),
    plane_orientation='x_axes', slice_index=10,)
mlab.pipeline.image_plane_widget(mlab.pipeline.scalar_field(scalar),
    plane_orientation='y_axes', slice_index=10,)
mlab.outline()
mlab.show()
```

Although we cannot show it, the user can interact with the visualization by moving the cuts and rotating the figures. And to finish, we place both contours and cut planes in the same plot to produce the interesting visualization as shown in Figure 1.13b.

1.6 Plotting Exercises

We encourage you to make your own plots and personalize them by trying out other commands and by including further options in the commands. The Matplotlib documentation is extensive and available on the Web. As an exercise, explore:

1. how to zoom in and zoom out on sections of a plot?
2. how to save your plots to files in various formats?
3. how to print up your graphs?
4. the options available from the pull-down menus?
5. how to increase the space between subplots?
6. and how to rotate and scale the surfaces.

1.7 Python's Algebraic Tools

While this book's focus is on the use of Python for numerical simulations, this is not to discount the importance of computational symbolic manipulations (even though that may be the way we feel). Python actually has (at least) two packages that can be used for symbolic manipulations, and they are quite different. As indicated in Section 1.4.3, the *Sage* package is very much in the same class as Maple and Mathematica, with a notebook graphical interface that lets the user create publication quality text, within which Python programs can be run, or the equations can be manipulated symbolically. Yet Sage is a big and powerful package that goes beyond pure Python by including multiple computer algebra systems as well as visualization tools and more. Using the multiple features of Sage can get to be quite complicated, and, in fact, books have been written and workshops taught on the use of Sage. We refer the interested reader to the online Sage documentation page at www.sagemath.org/help.html.

The *SymPy* package for symbolic manipulations runs very much like any other Python package from within your regular Python shell. It can be downloaded from https://github.com/sympy/sympy/releases, or you can use the Canopy distribution that includes SymPy. Now we give some simple examples of SymPy's use, but you really need to start with the *SymPy Tutorial* http://docs.sympy.org/latest/tutorial/ if you want to use SymPy. (Note, despite the fact that we are working within a Python shell, SymPy has automatically found our LaTeXapplication and used it to format the output.) To start, we will take some derivatives to show that SymPy knows calculus:

```
>>> from sympy import *
>>> x, y = symbols('x y')
>>> y = diff(tan(x),x); y tan^2(x)+1
>>> y = diff(5*x**4 + 7*x**2, x, 1); y              # dy/dx with optional 1
    20x^3 + 14x
>>> y = diff(5*x**4+7*x**2, x, 2); y                # d^2y/dx^2
    2(30x^2 + 7)
```

We see here that we must first import methods from SymPy and then use the `symbols` command to declare the variables x and y as algebraic. The rest is rather obvious, with `diff` being the derivative operator and the `x` argument in `diff` indicating what we are taking the derivative with respect to x. Now let us try expansions:

```
>>> from sympy import *
>>> x, y = symbols('x y')
>>> z = (x + y)**8; z
    (x + y)^8
>>> expand(z)  x^8 + 8x^7y + 28x^6y^2 + 56x^5y^3 + 70x^4y^4 + 56x^3y^5 + 28x^2y^6 + 8xy^7 + y^8
```

SymPy knows about infinite series and different expansion points:

```
>>> sin(x).series(x, 0)                            # Usual sin x series about 0
     x - x^3/6 + x^5/120 + O(x^6)
>>> sin(x).series(x,10)                            # sin x about x= 10
     sin(10) + x cos(10) - x^2 sin(10)/2 - x^3 cos(10)/6 + x^4 sin(10)/24 + x^5 cos(10)/120 + O(x^6)
```

```
>>> z = 1/cos(x); z                              # A division, not an inverse
    1/ cos(x)
>>> z.series(x, 0)                               # Expand 1/ cos x about x = 0
    1 + x^2/2 + 5x^4/24 + O(x^6)
```

One of the classic difficulties with computer algebra systems is that even if the answer is correct, if it is not simple, then it probably is not useful. And so, SymPy has a `simplify` function as well as a `factor` function (and `collect`, `cancel` and `apart` which we will not illustrate):

```
>>> factor(x**2 -1)
    (x − 1)(x + 1)                                        # A nice answer
>>> factor(x**3 - x**2 + x - 1)
    (x − 1)(x^2 + 1)
>>> simplify((x**3 + x**2 - x - 1)/(x**2 + 2*x + 1))
    x − 1                                                          # Much
         better!
>>> simplify(x**3+3*x**2*y+3*x*y**2+y**3)
    x^3 + 3x^2 y + 3x y^2 + y^3                   # No help!
>>> factor(x**3+3*x**2*y+3*x*y**2+y**3)
    (x + y)^3                                                 # Much better!
>>> simplify(1 + tan(x)**2)
    cos(x)^(−2)
>>> simplify(2*tan(x)/(1+tan(x)**2))
    sin(2x)
```