Lecture Notes for
Physics 801: Numerical Methods

Jolien Creighton

Fall 2016

Contents

1 Tutorial **1**
1.1 Using the terminal 1
1.2 Python 3
1.3 References 10

2 Ordinary differential equations **13**
2.1 Radioactive decay 13
2.2 Projectile motion 17
2.3 Pendulum 22
2.4 Orbital motion 30

3 Partial differential equations **47**
3.1 Waves 47
3.2 Heat diffusion 64
3.3 Schrödinger equation 73
3.4 Electric potentials 87

4 Random systems **101**
4.1 Random walks 101
4.2 Ising model 111
4.3 Variational method 119

5 Data reduction **125**
5.1 Statistical description of data 125
5.2 Statistical tests of data distribution 134
5.3 Data modelling 146
5.4 Bayesian inference 159

A Algorithms **169**
A.1 Linear algebra 169
A.2 Root finding 175
A.3 Minimization 178
A.4 Interpolation 184
A.5 Integration 185
A.6 Fourier transform 201

B Binary representation of numbers **211**

Index **216**

List of Listings

1.1 Program hello.py 3
1.2 Program cosine.py 7
1.3 Program plotcos.py 8
1.4 Program plottrig.py 9

2.1 Program decay.py 14
2.2 Program decayerr.py 15
2.3 Program projectile.py 18
2.4 Program shoot.py 20
2.5 Program pendulum.py 23
2.6 Modification to program pendulum.py 25
2.7 Program dblpend.py 28
2.8 Program planet.py 37
2.9 Program halley.py 41

3.1 Program waveftcs.py 52
3.2 Modified lines in program wavelax.py 55
3.3 Program leapfrog.py 61
3.4 Program heatftcs.py 67
3.5 Program implicit.py 70
3.6 Program oscillator.py 78
3.7 Modified lines in program tunnel.py 81
3.8 Program scatter.py 84
3.9 Program relax.py 90
3.10 Program overrelax.py 94
3.11 Program charge.py 97

4.1 Program randwalk.py 102
4.2 Program diffuse.py 109

4.3 Program `ising.py` 116
4.4 Program `variational.py` 122

5.1 Data file `cepheids.dat` 133
5.2 Program `cepheid.py` 133
5.3 Program `ks1.py` 136
5.4 Program `ks2.py` 138
5.5 Program `chisq1.py` 143
5.6 Program `chisq2.py` 144
5.7 Program `regress.py` 148
5.8 Program `svdfit.py` 156
5.9 Program `mcmc.py` 164

A.1 Module `linalg.py` 173
A.2 Module `root.py` 176
A.3 Module `minimize.py` 180
A.4 Program `traveler.py` 182
A.5 Module `interpolate.py` 185
A.6 Module `integrate.py` 196
A.7 Module `fft.py` 206

List of Figures

1.1 Screen-shot of plot produced by plotcos.py 10

2.1 Results from program decay.py 15
2.2 Results from program decay.py 16
2.3 Results from program projectile.py 19
2.4 Results from program shoot.py 22
2.5 Results from program pendulum.py 24
2.6 Results from the modified program pendulum.py 25
2.7 The double pendulum. 27
2.8 Results from program dblpend.py (regular motion) 30
2.9 Results from program dblpend.py (chaotic motion) 31
2.10 Eccentric anomaly 33
2.11 Results from program planet.py for the orbit of Mercury 39
2.12 Results from program halley.py 43
2.13 Initial configuration for the three body problem. 46

3.1 Results from program waveftcs.py 54
3.2 Results from running the program wavelax.py ($c\Delta t = \Delta x$) 56
3.3 Results from running the program wavelax.py ($c\Delta t = 1.1\Delta x$) 57
3.4 Results from running the program wavelax.py ($c\Delta t = 0.9\Delta x$) 58
3.5 Grid used for the leapfrog method 59
3.6 Results from running the program leapfrog.py ($c\Delta t = \Delta x$) 63
3.7 Results from running the program leapfrog.py ($c\Delta t = 0.5\Delta x$) . . . 64
3.8 Results from program implicit.py 72
3.9 Results from program oscillator.py 79
3.10 Results from program tunnel.py 82
3.11 Results from program scatter.py 85
3.12 Boundary value problem for Laplace's equation 88

3.13 Results from program `relax.py` . . . 90
3.14 Staggered lattice for successive over-relaxation . . . 93
3.15 Results from program `overrelax.py` . . . 95
3.16 Boundary value problem for Poisson's equation . . . 96
3.17 Results from program `charge.py` . . . 98
3.18 Boundary value problem for a capacitor . . . 100

4.1 Results from program `randwalk.py` for 3 walks . . . 103
4.2 Results from program `randwalk.py` for 500 walks . . . 104
4.3 Snapshots from program `diffuse.py` . . . 108
4.4 Entropy evolution from program `diffuse.py` . . . 109
4.5 Ising model energy from program `ising.py` . . . 115
4.6 Ising model energy from program `ising.py` . . . 116
4.7 Ising model energy from program `ising.py` . . . 117
4.8 Snapshots from program `variational.py` . . . 121
4.9 Energy functional results from program `variational.py` . . . 122

5.1 Results from program `cepheid.py` . . . 135
5.2 Results from program `ks1.py` . . . 138
5.3 Results from program `ks2.py` . . . 140
5.4 Results from program `chisq1.py` . . . 142
5.5 Results from program `chisq2.py` . . . 145
5.6 Results from program `regress.py` . . . 149
5.7 Results from program `svdfit.py` . . . 156
5.8 Error ellipses from program `svdfit.py` . . . 157
5.9 Data analyzed by program `mcmc.py` . . . 163
5.10 Evolution of parameters for program `mcmc.py` . . . 164
5.11 Posterior distribution from program `mcmc.py` . . . 165

A.1 Traveling salesperson problem . . . 182
A.2 Methods of integration . . . 188
A.3 Illustration of aliasing . . . 203

List of Tables

1.1 Common UNIX commands. 2

2.1 Cash-Karp parameters for `rk45`. 42

3.1 Hermite polynomials. 78

5.1 Cepheid variable data 133
5.2 Confidence level for $\Delta\chi^2$ ellipsoids 154

A.1 Legendre polynomials and their abscissas and weights. 195
A.2 Symmetries of the Fourier transform. 201
A.3 Properties of the Fourier transform. 202

List of Exercises

1.1 The cosine function . . . 10

2.1 Decay chain for two species of nuclei . . . 14
2.2 Projectile motion with drag . . . 18
2.3 Farmer trying to catch a pig . . . 21
2.4 Energy conservation of Cromer's method for the simple pendulum . . . 25
2.5 Normal modes and chaos of a double pendulum system . . . 30
2.6 Runge-Kutta method applied to the simple harmonic oscillator . . . 36
2.7 Perihelion advance of Mercury . . . 39
2.8 Three body dynamics . . . 45

3.1 Waves on a string . . . 63
3.2 Diffusion of a Gaussian distribution . . . 71
3.3 Quantum scattering off of a potential shelf . . . 82
3.4 Two-dimensional parallel plate capacitor . . . 99

4.1 Random walk in a vertical column . . . 110
4.2 First-order phase transitions in the Ising model . . . 119
4.3 Variational method of quantum mechanics . . . 124

5.1 Using Cepheid variables to determine distance of M101 . . . 150

A.1 Basin of convergence for Newton's method . . . 177

B.1 Loss of significance . . . 215

Chapter 1

Tutorial

This chapter is intended for those who are unfamiliar with UNIX or the Python programming language. This is not intended to be a very comprehensive tutorial, but I have identified some web pages where you can find more information.

1.1 Using the terminal

In this section I give a *very* brief introduction to common commands that you will use with the terminal.

The first challenge is to locate the terminal. On a Mac, you use the finder to go to the Applications folder, and then go to the Utilities folder. The terminal is called `Terminal`. You can also get it by searching for ‘Terminal’ in with Spotlight (the magnifying glass at the top right of the menu bar).

Once the terminal is open you’ll have a command line prompt. You can now type shell commands. Table 1.1 lists some of the more common UNIX commands and gives some examples of their usage.

The files and folders that you are used to dealing with can be accessed from the commands you type on the terminal (folders are called directories). For example, on a Mac, the commands

```
cd
cd Documents
ls
```

will show you all the files in your `Documents` folder. (The first command, `cd`, takes you to your home directory, the second command, `cd Documents` puts you into the directory `Documents`, and the third command `ls` lists all the files in that directory.)

That’s all the introduction to UNIX commands I am going to give here. There are plenty of resources on the web for learning more. One tutorial for beginners, for example, is `http://www.ee.surrey.ac.uk/Teaching/Unix/`.

`cat`	concatenate/print files	print the contents of `file1` to the screen:
		`cat file1`
		concatenate files `file1` and `file2` to `file3`:
		`cat file1 file2 > file3`

`cd`	change directory	change to home directory:
		`cd`
		change to subdirectory `Desktop`:
		`cd Desktop`
		change to parent directory:
		`cd ..`
		change to directory `/usr/bin`:
		`cd /usr/bin`

`cp`	copy files	make a copy of `file1` as `file2`:
		`cp file1 file2`
		put a copy of `file1` in directory `/tmp`:
		`cp file1 /tmp`

`ls`	list directory contents	list contents of current directory:
		`ls`
		list contents of directory /usr/bin:
		`ls /usr/bin`

`man`	online manual	display manual entry for the command `cat` `man cat`
`mkdir`	make directories	make new subdirectory `subdir`: `mkdir subdir`

`mv`	move files	rename `oldfile` as `newfile`:
		`mv oldfile newfile`
		move `file1` to directory `/tmp`:
		`mv file1 /tmp`

`pico`	file editor	edit the file `file1`: `pico file1`
`pwd`	return directory name	print working directory: `pwd`
`rm`	remove file	delete the file `file1`: `rm file1`

Table 1.1: Common UNIX commands.

1.2 Python

Now you need to learn how to use the Python language. One good resource is `http://www.learnpython.org/`. This section also contains a few simple example Python programs. Soon you'll want to consult the Python documentation, `http://docs.python.org/`. For scientific computing the packages Numpy and Scipy are very helpful; documentation for these are found at `http://docs.scipy.org/`. Finally, the package matplotlib, `http://matplotlib.org/`, is used for plotting.

Let's start with a very basic program. This program, `hello.py`, will print the words 'hello, world' to the screen. The file `hello.py` has just a single line. The listing (in full) is given here:

Listing 1.1: Program `hello.py`

```
1 print 'hello, world'
```

You can run it with the command `python hello.py`. The output you will get is

```
hello, world
```

The Python program is an interpreter so you just run it and enter Python commands from the Python command line prompt. This is a good way to try things out and to get to know the language. Just enter the command `python` and you will enter the interpreter in an interactive session. An example of such a session might be something like

```
Python 2.7.2 (default, Jan 10 2012, 15:06:11)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> print 'hello, world'
hello, world
>>>
```

Here the input is indicated with boldface text. To exit the interpreter you type `^D` (Control-D). Within the interpreter, the command `help()` will enter an interactive help session where you can get more information.

A somewhat more convenient program to use for interactive python sessions is called `ipython`. It has many useful and helpful features which you can read about at `http://ipython.org/`. If you run the program as `ipython -pylab` then you will have access to all of the functions in the `pylab` module (see below); the resulting programming environment is then similar to MATLAB.

We will give a very brief overview of the important aspects of the Python programming language. This overview is in no way complete; rather it is intended to give an introduction to Python programming by way of example.

Expressions and assignment There are basic mathematical operations for addition, `+`, subtraction, `-`, multiplication, `*`, and division, `/`, along with the assignment operation, `=`. For example,

```
>>> x = 3
>>> y = x * 4 + 5
>>> print x
3
>>> print y
17
```

Some other basic operations include the power operator, `**`, floor-division, `//`, and the modulo operator, `%`. Parentheses can be used to ensure that operations are performed in the desired order. Here is an illustration:

```
>>> x = 3.14
>>> y = 0.7
>>> print x ** y
2.22766947371
>>> print x // y
4.0
>>> print x % y
0.34
>>> print (x//y)*y + (x%y)
3.14
```

An operation and an assignment can be combined with assignment operators such as `+=`, `-=`, `*=`, and `/=`. For example, the statement `i = i + 1` is the same as `i += 1`.

Data types Some of the basic data types in Python are integers, floating-point numbers, complex numbers, strings, and lists. We've already seen some of these. A string is indicated by an open quotation mark, either `'` or `"`, and continues until a close quotation mark. Strings can be "added" together, which forms a concatenation. For example:

```
>>> x = 'hello'
>>> y = 'world'
>>> print x + ', ' + y
hello, world
```

A list is a collection of elements contained between an open bracket, `[`, and a close bracket, `]`, and separated by commas, `,`. The elements of a list can be strings, integers, floating-point numbers, etc. The `len` operator will tell you the number of elements in a list. Like strings, lists can be added, which appends one list to another. A list can also be multiplied by a number, which repeats the list that number of times. Finally, the individual elements of a list can be accessed as seen in the example below:

```
>>> a = [1, 2, 3]
>>> b = [4] * 3
>>> print a
[1, 2, 3]
```

```
>>> print b
[4, 4, 4]
>>> print a + b
[1, 2, 3, 4, 4, 4]
>>> print len(a + b)
6
>>> print a[0]
1
>>> print a[1]
2
```

Notice that the index `0` refers to the first element in the list, `1` to the next element, and so on. Incidentally, the index `-1` refers to the last element in the list. A useful way of generating a list of numbers is with the `range` function:

```
>>> a = range(10)
>>> print a
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Complex numbers are obtained by multiplying a floating-point number by the complex imaginary constant `1j`, for example:

```
>>> x = 4
>>> y = 3
>>> z = x + 1j * y
>>> print z
(4+3j)
>>> print z.real
4.0
>>> print z.imag
3.0
>>> print abs(z)
5.0
```

Notice that the real and imaginary parts of a complex variable can be accessed as `z.real` and `z.imag`, while the magnitude of the complex variable can be obtained by `abs(z)`.

Conditionals Conditionals are an example of a compound statement. A group of statements are executed according to the boolean value of a test. In Python, the group of statements are indicated by a certain level of indentation. For example:

```
>>> x = 3
>>> y = 4
>>> if x == y:
...     print 'x is equal to y'
... elif x > y:
...     print 'the larger value is x'
...     print x
```

```
... else:
...     print 'the larger value is y'
...     print y
...
the larger value is y
4
```

The first indented block of code is only evaluated if `x` is equal to `y`, that is, if the boolean statement `x == y` is true; the second block of code is only executed if `x` is greater than `y`, that is, if the boolean statement `x > y`; and the third block of code after the `else:` statement is executed otherwise.

Loops A block of code can be executed repeatedly until some condition is satisfied in a loop. The most basic loop is the while-loop:

```
>>> i = 0
>>> while i < 10:
...     print i,
...     i = i + 1
...
0 1 2 3 4 5 6 7 8 9
```

However, for-loops are more common. In a for-loop, a variable is set to sequential values in, for example, a list of values. The equivalent for-loop would be

```
>>> for i in range(10):
...     print i,
...
0 1 2 3 4 5 6 7 8 9
```

Here is another example:

```
>>> colors = ['red', 'green', 'blue']
>>> for color in colors:
...     print 'my favorite color is ' + color
...
my favorite color is red
my favorite color is green
my favorite color is blue
```

List comprehensions In Python, lists can be created using a technique called list comprehension, an illustration of which is given in the following example which generates a list of the first ten powers of two:

```
>>> a = [2**n for n in range(10)]
>>> print a
[1, 2, 4, 8, 16, 32, 64, 128, 256, 512]
```

The function `sum` can be used to compute the sum of the elements in a list. For example, to compute the sum of squares of integers less than 10,

```
>>> s = sum([x**2 for x in range(10)])
>>> print s
285
```

but this can also be done slightly more concisely without actually creating the list:

```
>>> s = sum(x**2 for x in range(10))
>>> print s
285
```

Functions We have already seen some built-in functions, such as `range` and `abs`, and we have seen how additional functions can be imported from modules, such as the `sin` function in the `math` module. You can also create your own functions. Here is a simple example of a factorial function:

```
>>> def factorial(n):
...     ans = 1
...     while n > 1:
...             ans *= n
...             n -= 1
...     return ans
...
>>> for n in range(6):
...     print n, 'factorial is', factorial(n)
...
0 factorial is 1
1 factorial is 1
2 factorial is 2
3 factorial is 6
4 factorial is 24
5 factorial is 120
```

Having discussed some of the basic elements of Python programming, we can consider a more complicated program. This program, `cosine.py`, prints out the values of the function $\cos\theta$ for θ in the range $0 \leq \theta < 20$ radians with steps of $\Delta\theta = 0.1$ radians.

Listing 1.2: Program `cosine.py`

```
1  import math
2
3  # set values for the parameters
4  nsteps = 200
5  dtheta = 0.1
6
7  # loop incrementing i from 0 to nsteps - 1
```

```
8  for i in range(nsteps):
9      theta = i*dtheta
10     print theta, math.cos(theta)
```

Line 2 of this program imports the `math` module, which is needed for the cosine function. Then lines 9–11 are a loop in which the variable `i` is incremented from 0 to 199: The function `range(n)` returns the list `[0, 1, 2, ..., n-1]`, and with each iteration of the loop the variable `i` takes on the next value of this list. Line 11 contains the actual call to the cosine function: because it is contained in the module `math`, the function call is `math.cos(theta)`.

Now to run the program. The command `python cosine.py > cosine.out` redirects the output to the file `cosine.out`. The command `head cosine.out` will show you the first few lines

```
0.0 1.0
0.1 0.995004165278
0.2 0.980066577841
0.3 0.955336489126
0.4 0.921060994003
0.5 0.87758256189
0.6 0.82533561491
0.7 0.764842187284
0.8 0.696706709347
0.9 0.621609968271
```

(the first column is the value of θ and the second column is the value of $\cos\theta$) while the command `tail cosine.out` shows you the last few lines

```
19.0 0.988704618187
19.1 0.968802459407
19.2 0.939220346697
19.3 0.900253854747
19.4 0.852292323865
19.5 0.795814969814
19.6 0.731386095645
19.7 0.659649453373
19.8 0.581321811814
19.9 0.497185794871
```

What if we want to *plot* the output? The program `plotcos.py` is a modification that produces a plot of the cosine function.

Listing 1.3: Program `plotcos.py`

```
1  import math, pylab
2
3  nsteps = 200
4  dtheta = 0.1
5
6  # create lists of length nsteps for the values of theta and cosine
```

```
7  # initially the values are set to zero
8  theta = [0.0]*nsteps
9  cosine = [0.0]*nsteps
10
11 # loop to fill in the values of theta and cosine
12 for i in range(nsteps):
13     theta[i] = i*dtheta
14     cosine[i] = math.cos(theta[i])
15
16 # show a plot
17 pylab.plot(theta, cosine)
18 pylab.show()
```

When this program is run, a plot will pop up on the screen that looks like Fig. 1.1. There are several differences here. First, on line 2 the program imports the module `pylab` which contains (among other things) the functions required to make plots. Second, the program creates two lists named `theta` and `cosine`, each with `nsteps` elements: this is done on lines 9 and 10. (Note that the Python command `[0.0] * 3` produces `[0.0, 0.0, 0.0]`.) In lines 14 and 15 the elements of these lists are assigned to the correct values. Finally, line 18 produces the plot and line 19 displays it on the screen.

There are many ways to embellish the plot (e.g., adding a grid, axis labels, a title, a legend, etc.) and we'll see examples of these in the example programs to follow. The best way to learn a programming language is to experiment with it by writing your own programs and by looking at other programs to figure out what they do. Here is a somewhat fancier program that introduces a few more features that we have discussed. Try it.

Listing 1.4: Program `plottrig.py`

```
1  import math, pylab
2
3  # input the parameters
4  nsteps = input('enter number of steps -> ')
5  dtheta = input('enter step size (rad) -> ')
6
7  # these are 'list comprehensions'
8  theta = [i*dtheta for i in range(nsteps)]
9  sine = [math.sin(x) for x in theta]
10 cosine = [math.cos(x) for x in theta]
11 pylab.plot(theta, sine, 'o-b', label='sin')  # blue circle symbols with line
12 pylab.plot(theta, cosine, 'o-r', label='cos')  # red circle symbols with line
13 pylab.xlabel('theta (rad)')
14 pylab.ylabel('sin(theta) and cos(theta)')
15 pylab.legend()
16 pylab.title('the sine and cosine functions')
17 pylab.grid()
18 pylab.show()
```

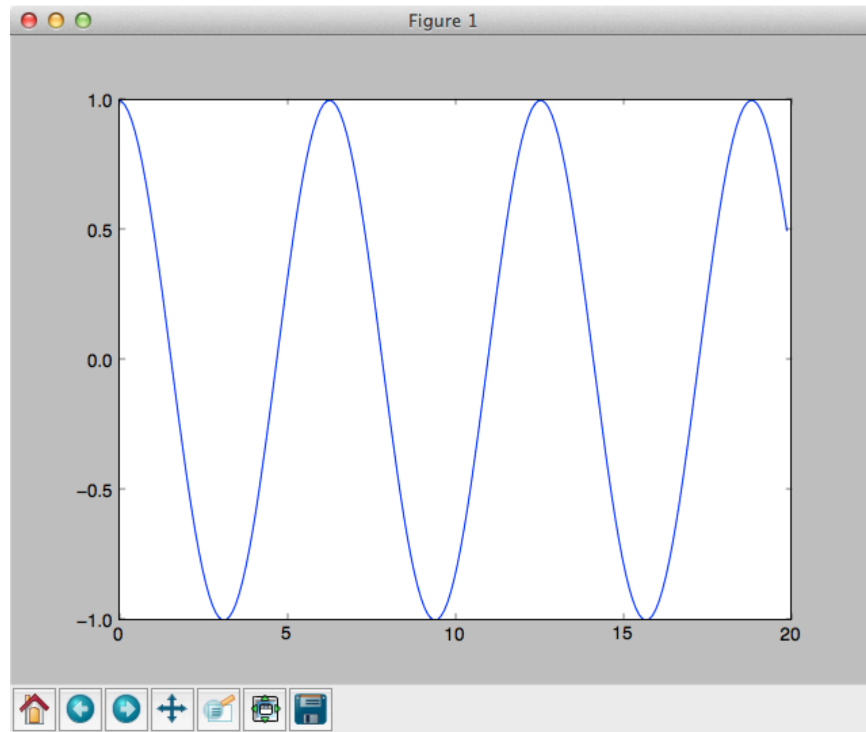

Figure 1.1: A screen-shot of the plot produced by the program `plotcos.py` (Listing 1.3).

Exercise 1.1 Produce a program that is similar to `plotcos.py` but do not use the `cos` function from the `math` module (or any other module). Instead, compute it using ordinary multiplication and additions. Consult some of the references in the next section if you need help finding an algorithm for computing the cosine function.

1.3 References

These notes are being compiled from material from a variety of sources. In particular, I am drawing heavily from the following references:

- *Computational Physics* (second edition) by Nicholas J. Giordano and Hisao Nakanishi (Pearson Prentice Hall, 2006) ISBN 0-13-146990-8.
- *Computational Physics* (revised and expanded) by Mark Newmann (2013) ISBN 978-148014551-1.

- *Numerical Recipes: The Art of Scientific Computing* (third edition) by William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery (Cambridge University Press, 2007) ISBN 978-0-521-88068-8.
- *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables* by Milton Abramowitz and Irene Stegun (Dover, 1964) ISBN 0-486-61272-4.

Chapter 2

Ordinary differential equations

2.1 Radioactive decay

Consider the radioactive decay of nuclei. The number of nuclei, N, follows the ordinary differential equation

$$\frac{dN}{dt} = -\frac{N}{\tau} \tag{2.1}$$

where τ is the decay time constant. This equation can be integrated directly, with the solution

$$N(t) = N_0 \mathrm{e}^{-t/\tau} \tag{2.2}$$

but we want to attempt to solve the equation numerically.

The straightforward approach is to express the number of nuclei at time $t + \Delta t$ in terms of the number at time t as

$$N(t + \Delta t) = N(t) + \frac{dN(t)}{dt}\,\Delta t + O(\Delta t^2). \tag{2.3}$$

Combining this equation with Eq. (2.1) we have

$$N(t + \Delta t) = N(t) - \frac{N(t)}{\tau}\,\Delta t + O(\Delta t^2). \tag{2.4}$$

If we start with N_0 nuclei at time $t = 0$, then at $t = \Delta t$ we will have $N(\Delta t) \approx N_0 - (N_0/\tau)\,\Delta t$; at $t = 2\Delta t$ we will have $N(2\Delta t) \approx N(\Delta t) - [N(\Delta t)/\tau]\,\Delta t$, and so on. The *truncation error* is $O(\Delta t^2)$ so if the step size Δt is small then we expect that our numerical solution should be close to the true solution. This method of integration of an ordinary differential equation is known as *Euler's method*.

Here is a program that will implement this method of integrating the differential equation for radioactive decay:

Listing 2.1: Program decay.py

```
import pylab

nuclei0 = input('initial number of nuclei -> ')
tau = input('decay time constant -> ')
dt = input('time step -> ')
tmax = input('time to end of simulation -> ')
nsteps = int(tmax/dt)
nuclei = [0.0]*nsteps
t = [0.0]*nsteps

# use Euler's method to integrate equation for radioactive decay
t[0] = 0.0
nuclei[0] = nuclei0
for i in range(nsteps-1):
    t[i+1] = t[i]+dt
    nuclei[i+1] = nuclei[i]-nuclei[i]/tau*dt
pylab.plot(t, nuclei, 'o-b')
pylab.xlabel('time')
pylab.ylabel('nuclei')
pylab.title('radioactive decay')
pylab.grid()
pylab.show()
```

The program requests the initial number of nuclei, N_0, the decay time constant τ, the time step Δt, and the total duration of the integration t_{max}. When this program is run with the input values $N_0 = 100$, $\tau = 1$, $\Delta t = 0.04$, and $t_{\text{max}} = 5$, the program produces the plot shown in Fig. 2.1.

Exercise 2.1 Write a program that computes the radioactive decay of *two* types of nuclei, A and B, where A nuclei decay into B nuclei. The system of differential equations is

$$\frac{dN_\text{A}}{dt} = -\frac{N_\text{A}}{\tau_\text{A}} \tag{2.5a}$$

$$\frac{dN_\text{B}}{dt} = \frac{N_\text{A}}{\tau_\text{A}} - \frac{N_\text{B}}{\tau_\text{B}} \tag{2.5b}$$

where τ_A and τ_B are the decay time constants for type A and type B nuclei respectively. Investigate how the behavior of $N_\text{A}(t)$ and $N_\text{B}(t)$ at early times and at late times for various values of the ratio $\tau_\text{A}/\tau_\text{B}$. Obtain analytic solutions for $N_\text{A}(t)$ and $N_\text{B}(t)$ and compare your numerical results to these solutions.

Let's now investigate how close to the exact answer our program is. Presumably when the step size Δt is large the error will be worse; also presumably errors grow with time. To see this, consider a modified version of our program decay.py which

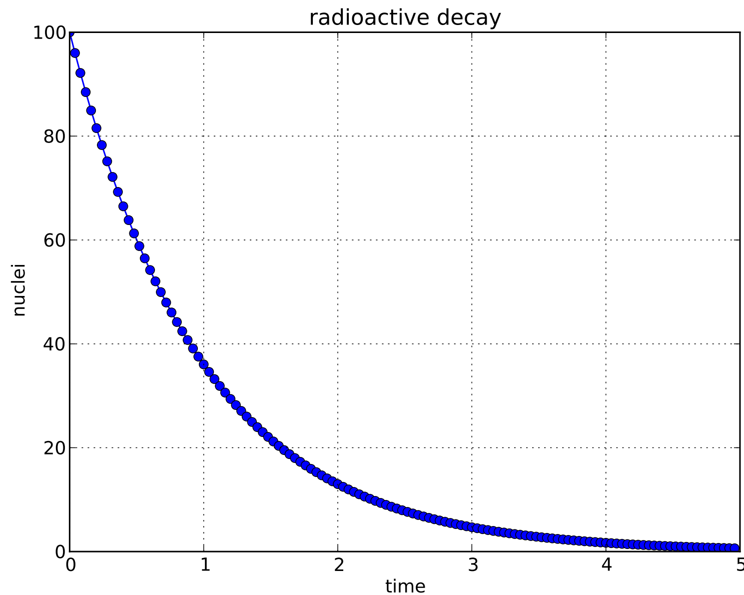

Figure 2.1: Results from running the program `decay.py` (Listing 2.1) with input $N_0 = 100$, $\tau = 1$, $\Delta t = 0.04$, and $t_{\max} = 5$.

plots the fractional difference between the numerical result and the exact result given by Eq. (2.2). Our new program will perform numerical evolutions at a number of different values of the step size so that we can see how the error depends on the degree of refinement of Δt.

Listing 2.2: Program `decayerr.py`

```
1  import math, pylab
2
3  nuclei0 = input('initial number of nuclei -> ')
4  tau = input('decay time constant -> ')
5  dtlow = input('lowest resolution time step -> ')
6  nres = input('number of resolution refinements -> ')
7  tmax = input('time to end of simulation -> ')
8  for n in range(nres):
9      refine = 10**n
10     dt = dtlow/refine
11     nsteps = int(tmax/dt)
12     nuclei = nuclei0
13     err = [0.0]*nsteps
14     t = [0.0]*nsteps
15
16     # use Euler's method to integrate equation for radioactive decay compute
```

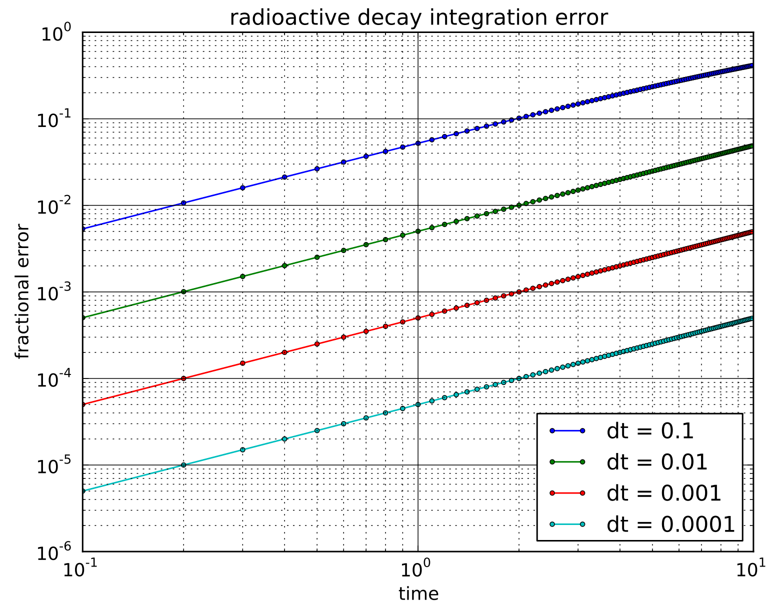

Figure 2.2: Results from running the program `decay.py` (Listing 2.2) with input $N_0 = 100$, $\tau = 1$, $\Delta t = 0.1$, $N_{\mathrm{res}} = 4$, and $t_{\mathrm{max}} = 10$.

```
17        # error relative to exact solution
18        for i in range(nsteps-1):
19            t[i+1] = t[i]+dt
20            nuclei = nuclei-nuclei/tau*dt
21            exact = nuclei0*math.exp(-t[i+1]/tau)
22            err[i+1] = abs((nuclei-exact)/exact)
23
24        # plot the error at this resolution
25        pylab.loglog(t[refine::refine], err[refine::refine], '.-', label='dt
             = '
26                     +str(dt))
27    pylab.legend(loc=4)
28    pylab.xlabel('time')
29    pylab.ylabel('fractional error')
30    pylab.title('radioactive decay integration error')
31    pylab.grid(linestyle='-', which='major')
32    pylab.grid(which='minor')
33    pylab.show()
```

This program produces the results shown in Figure 2.2.

The errors grow close to linearly with time (the lines in the log-plot have approximately unit slope) and each factor of 10 in refinement decreases the fractional

error by a factor of 10. To understand this, note that the term that we threw away in the Taylor expansion of our ordinary differential equation was the d^2N/dt^2 term, so each step introduces an error of

$$e_i \approx \frac{1}{2}\frac{d^2N(t_i)}{dt^2}\,\Delta t^2 = \frac{N(t_i)}{2\tau^2}\,\Delta t^2. \tag{2.6}$$

This is known as the *local error*. If the local error of a numerical integration scheme is $O(\Delta t^{p+1})$ as $\Delta t \to 0$ then we say it is *order* p. Euler's method is therefore a first order integration scheme. The *global error* is the error accumulated when the integration is performed for some duration T. The number of steps required is $n = T/\Delta t$, and each step $i = 1 \ldots n$ accumulates an error e_i, so we would expect the global error to be

$$E_n \le \sum_{i=1}^{n} e_i \le T\,\frac{N_0}{2\tau^2}\,\Delta t \tag{2.7}$$

since $e_i \le [N_0/(2\tau^2)]\,\Delta t^2$. Note that for an order p integration scheme, the global error will be $O(\Delta t^p)$; furthermore, the error grows with time T. For the Euler method, the error grows approximately linearly with T and with Δt, which is what we see in Fig. 2.2.

The Euler method is not a recommended method for solving ordinary differential equations. Being merely first order, a desired accuracy is achieved only for very small values of Δt, and therefore many integration steps are required in order to evolve the system for a given duration T. But the computational cost of Euler's method is not its only shortfall: it is not particularly stable either, as we will see in the Sec. 2.3.

2.2 Projectile motion

Another example ordinary differential equation is that of projectile motion, for which the equations of motion are

$$\frac{dx}{dt} = v_x, \qquad \frac{dv_x}{dt} = 0, \tag{2.8a}$$

$$\frac{dy}{dt} = v_y, \qquad \frac{dv_y}{dt} = -g, \tag{2.8b}$$

where g is the acceleration due to gravity. We can use the Euler method to write each derivative in a finite difference form that is suitable for numerical integration:

$$x_{i+1} = x_i + v_{x,i}\Delta t, \qquad v_{x,i+1} = v_{x,i}, \tag{2.9a}$$

$$y_{i+1} = y_i + v_{y,i}\Delta t, \qquad v_{y,i+1} = v_{y,i} - g\Delta t. \tag{2.9b}$$

The trajectories of a projectile launched with velocity $v_0 = 10\,\mathrm{m\,s^{-1}}$ at various angles are plotted by the program `projectile.py` and are plotted in Fig. 2.3.

Listing 2.3: Program `projectile.py`

```
import math, pylab

g = 9.8  # standard freefall (m/s^2)
v0 = 10.0  # initial velocity (m/s)
angles = [30.0, 35.0, 40.0, 45.0, 50.0, 55.0] # launch angles (degrees)
dt = 0.01  # time step (s)
for theta in angles:
    x = [0.0]
    y = [0.0]
    vx = [v0*math.cos(theta*math.pi/180.0)]
    vy = [v0*math.sin(theta*math.pi/180.0)]

    # use Euler's method to integrate projectile equations of motion
    i = 0
    while y[i] >= 0.0:

        # extend the lists by appending another point
        x += [0.0]
        y += [0.0]
        vx += [0.0]
        vy += [0.0]

        # apply finite difference approx to equations of motion
        x[i+1] = x[i]+vx[i]*dt
        y[i+1] = y[i]+vy[i]*dt
        vx[i+1] = vx[i]
        vy[i+1] = vy[i]-g*dt
        i = i+1

    # plot the trajectory
    pylab.plot(x, y, label=str(theta)+' degrees')

pylab.title('trajectory of a projectile')
pylab.xlabel('x (m)')
pylab.ylabel('y (m)')
pylab.ylim(ymin=0.0)
pylab.legend()
pylab.show()
```

We see, as expected, that the greatest range is achieved for a launch angle of 45°.

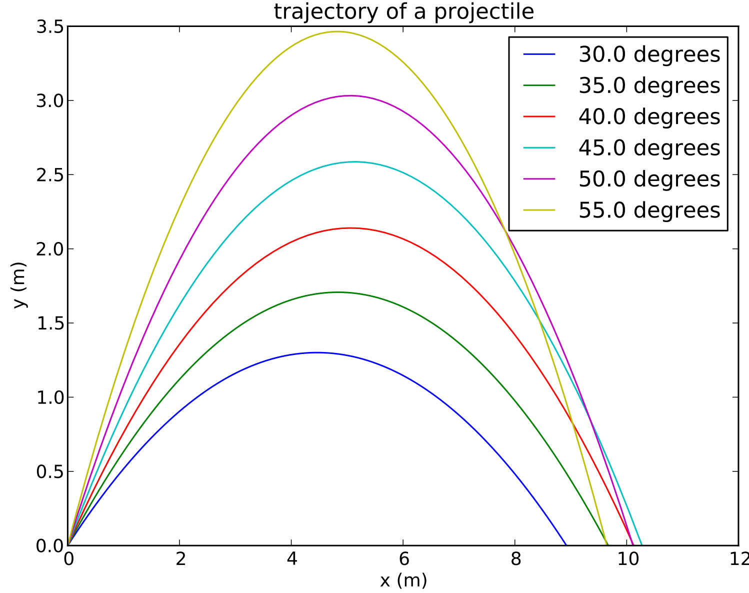

Figure 2.3: Results from running the program `projectile.py` (Listing 2.3). It is seen that the greatest range is achieved with a launch angle of $\theta = 45°$.

Exercise 2.2 When air friction is included there is an additional drag force that is predominantly proportional to the velocity of the projectile squared; the equations of motion become

$$\frac{dx}{dt} = v_x, \qquad \frac{dv_x}{dt} = -bvv_x, \tag{2.10a}$$

$$\frac{dy}{dt} = v_y, \qquad \frac{dv_y}{dt} = -g - bvv_y, \tag{2.10b}$$

where b is a drag constant and $v = \sqrt{v_x^2 + v_y^2}$ is the velocity of the projectile. Modify the program `projectile.py` to include a drag force and plot the trajectories for various launch angles when $b = 0.04\,\mathrm{m}^{-1}$. Find the launch angle that has the greatest range, $\theta_{\max}$, for fixed launch velocity $v_0 = 10\,\mathrm{m\,s}^{-1}$. How does $\theta_{\max}$ vary with b for $0\,\mathrm{m}^{-1} \leq b \leq 0.1\,\mathrm{m}^{-1}$?

Finding the trajectory of a projectile given its initial conditions, $v_{x,0}$ and $v_{y,0}$ or equivalently v_0 and θ, is relatively simple. However, suppose that we want to find the launch angle θ required to hit a target at a given distance with a given initial velocity v_0. This is an example of a *two point boundary value problem*. One approach to solving such a problem is known (aptly) as the *shooting method*.

The idea is straightforward: make a guess at the value of θ, perform the integration, determine how much you miss your mark, and then iteratively refine your guess until you are close enough to the target. If $\Delta x(\theta)$ is the amount that you miss the target with the launch angle θ then the goal is to solve the equation

$$\Delta x(\theta) = 0 \tag{2.11}$$

for θ. This general problem is called *root finding*. Here we'll employ a rather simple method for solving for a root known as the *bisection method*. Suppose that we know that the root of Eq. (2.11) lies somewhere in the range $\theta_1 < \theta < \theta_2$ and $\Delta x(\theta_1)$ has the opposite sign of $\Delta x(\theta_2)$ (that is, if $\Delta x(\theta_1) < 0$ then $\Delta x(\theta_2) > 0$, or vice versa). Then we say that θ_1 and θ_2 *bracket* the root. We begin by evaluating $\Delta x(\theta_1)$, $\Delta x(\theta_2)$, and $\Delta x(\theta_{\text{guess}})$ with θ_{guess} midway between θ_1 and θ_2, $\theta_{\text{guess}} = \frac{1}{2}(\theta_1 + \theta_2)$. If the sign of $\Delta x(\theta_{\text{guess}})$ is the same as the sign of $\Delta x(\theta_1)$ then we know that the root must be between θ_{guess} and θ_2, so we assign θ_1 to θ_{guess}, and make a new guess midway between the new θ_1 and θ_2. Otherwise, if the sign of $\Delta x(\theta_{\text{guess}})$ is the same as the sign of $\Delta x(\theta_2)$ then we know that the root must be between θ_1 and θ_{guess}, so we assign θ_2 to θ_{guess}, and make a new guess midway between θ_1 and the new θ_2. We continue this iteration until we are "close enough," i.e., $|\Delta x(\theta_{\text{guess}})| < \epsilon$ for some small value of ϵ.

For the problem at hand, let the target be located at a distance x_{target} and let the point where the projectile hits the ground when launched at angle θ be $x_{\text{ground}}(\theta)$. Define $\Delta x(\theta) = x_{\text{ground}}(\theta) - x_{\text{target}}$ so that $\Delta x(\theta) > 0$ if we've shot too far and $\Delta x(\theta) < 0$ if we've shot too near. Then, if $0 < x_{\text{target}} < x_{\text{max}}$ where we know $x_{\text{ground}}(0^\circ) = 0$ and $x_{\text{ground}}(45^\circ) = x_{\text{max}}$, then we know $\theta_1 = 0^\circ$ and $\theta_2 = 45^\circ$ bracket the root. The program `shoot.py` uses the shooting method to compute the trajectory of a projectile that is launched from $x = 0$ with a fixed velocity and lands at point $x = x_{\text{ground}}$. The results from this program run with an initial velocity $v_0 = 10\,\text{m}\,\text{s}^{-1}$ and target location $x_{\text{target}} = 8\,\text{m}$ are shown in Fig. 2.4.

Listing 2.4: Program `shoot.py`

```
1  import math, pylab
2
3  g = 9.8  # standard freefall (m/s^2)
4  v0 = input('initial velocity (m/s) -> ')
5  xtarget = input('target range (m) -> ')
6  eps = 0.01  # how close we must get (m)
7  dt = 0.001  # time step (s)
8  theta1 = 0.0  # bracketing angle (degrees) that falls too short
9  theta2 = 45.0  # bracketing angle (degrees) that falls too far
10 dx = 2*eps  # some initial value > eps
11 while abs(dx) > eps:
12     # guess at the value of theta
13     theta = (theta1+theta2)/2.0
14     x = [0.0]
15     y = [0.0]
16     vx = [v0*math.cos(theta*math.pi/180.0)]
```

```
    vy = [v0*math.sin(theta*math.pi/180.0)]

    # use Euler's method to integrate projectile equations of motion
    i = 0
    while y[i] >= 0.0:

        # apply finite difference approx to equations of motion
        x += [x[i]+vx[i]*dt]
        y += [y[i]+vy[i]*dt]
        vx += [vx[i]]
        vy += [vy[i]-g*dt]
        i = i+1

    # we hit the ground somewhere between step i-1 and i interpolate to find
    # this location
    xground = x[i-1]+y[i-1]*(x[i]-x[i-1])/(y[i]-y[i-1])

    # update the bounds bracketing the root
    dx = xground-xtarget
    if dx < 0.0:  # too short: update smaller angle
        theta1 = theta
    else:  # too far: update larger angle
        theta2 = theta

# plot the correct trajectory
pylab.plot(x, y)
pylab.plot([xtarget], [0.0], 'o')
pylab.annotate('target', xy=(xtarget, 0), xycoords='data', xytext=(5, 5),
               textcoords='offset points')
pylab.title('trajectory of a projectile with theta = %.2f degrees'%theta)
pylab.xlabel('x (m)')
pylab.ylabel('y (m)')
pylab.ylim(ymin=0.0)
pylab.show()
```

Exercise 2.3 A pig and a farmer are both in a square pigpen. The farmer is in the southeast corner while the pig is in the southwest corner. In the northwest corner (directly north of the pig) is an open gate. The pig runs toward the gate at a constant speed. The farmer chases the pig, also at constant speed, but always so that the farmer is running directly toward where the pig is at that instant. Numerically determine how many times faster than the pig the farmer must be running in order to just catch the pig before it escapes.

If you can derive an exact closed-form solution to this problem, compare the exact solution to the numerical solution that you obtained.

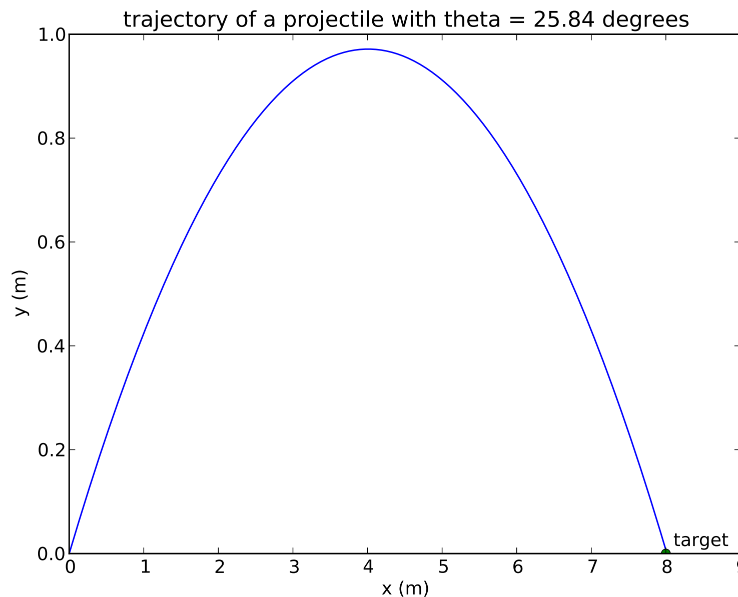

Figure 2.4: Results from running the program `shoot.py` (Listing 2.4) with initial velocity $v_0 = 10\,\mathrm{m\,s^{-1}}$ and target location $x_{\text{target}} = 8\,\mathrm{m}$. The angle required to hit the target is $\theta = 25.84°$.

2.3 Pendulum

A pendulum of length ℓ has the equation of motion

$$\frac{d^2\theta}{dt^2} = -\frac{g}{\ell}\sin\theta. \tag{2.12}$$

For small amplitudes of oscillation, $\theta \ll 1$, we can make the approximation $\sin\theta \approx \theta$, and the ordinary differential equation becomes one of a simple harmonic oscillator:

$$\frac{d^2\theta}{dt^2} = -\frac{g}{\ell}\theta. \tag{2.13}$$

The solution to this equation is simply

$$\theta(t) = \theta_0 \cos\omega_0 t \tag{2.14}$$

where $\omega_0^2 = g/\ell$ and we have assumed the pendulum starts from rest with an initial displacement θ_0.

To obtain a numerical solution, the second order ordinary differential equation

is converted to a coupled system of first order ordinary differential equations

$$\frac{d\omega}{dt} = -\frac{g}{\ell}\,\theta \tag{2.15a}$$

$$\frac{d\theta}{dt} = \omega. \tag{2.15b}$$

Using Euler's method these are written in discrete form as

$$\omega_{i+1} = \omega_i - \frac{g}{\ell}\theta_i \Delta t \tag{2.16a}$$

$$\theta_{i+1} = \theta_i + \omega_i \Delta t \tag{2.16b}$$

where the ith value corresponds to $t = i\,\Delta t$. The program `pendulum.py` is an implementation of Eqs. (2.16a) and (2.16b).

Listing 2.5: Program `pendulum.py`

```
import pylab

g = 9.8  # standard freefall (m/s^2)
l = input('pendulum length (meters) -> ')
theta0 = input('initial angle (radians) -> ')
dt = input('time step (seconds) -> ')
tmax = input('time to end of simulation (seconds) -> ')
nsteps = int(tmax/dt)
omega = [0.0]*nsteps
theta = [0.0]*nsteps
t = [0.0]*nsteps

# use Euler's method to integrate pendulum equations of motion
theta[0] = theta0
for i in range(nsteps-1):
    omega[i+1] = omega[i]-g/l*theta[i]*dt
    theta[i+1] = theta[i]+omega[i]*dt
    t[i+1] = t[i]+dt
pylab.plot(t, theta)
pylab.xlabel('time (s)')
pylab.ylabel('theta (rad)')
pylab.title('simple pendulum (Euler method)')
pylab.show()
```

If we run this program with values $\ell = 1\,\mathrm{m}$, $\theta_0 = 0.2\,\mathrm{rad}$, $\Delta t = 0.04\,\mathrm{s}$, and $t_{\max} = 10\,\mathrm{s}$, we obtain the results shown in Fig. 2.5. Clearly something is wrong!

What is going wrong? The evolution is unstable: the amplitude should remain constant, but instead it is increasing. The problem is that the Euler method is *unstable*. To see this, consider the total energy of the pendulum, which, in the small angle limit, is simply

$$E = \frac{1}{2}\,m\ell^2\omega^2 + \frac{1}{2}\,mg\ell\theta^2 \tag{2.17}$$

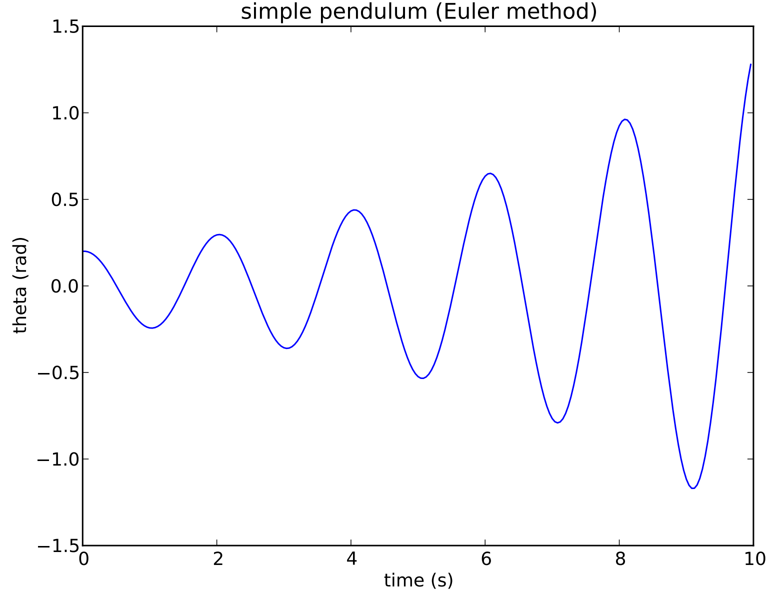

Figure 2.5: Results from running the program `pendulum.py` (Listing 2.5) with input $\ell = 1\,\mathrm{m}$, $\theta_0 = 0.2\,\mathrm{rad}$, $\Delta t = 0.04\,\mathrm{s}$, and $t_{\mathrm{max}} = 10\,\mathrm{s}$.

where m is the mass of the pendulum bob. To see how the energy will evolve in time, we use Eqs. (2.16a) and (2.16b) to obtain an equation relating the energy at step $i+1$, E_{i+1}, to the energy at the previous step i, E_i:

$$\begin{aligned} E_{i+1} &= \frac{1}{2} m\ell^2\omega_{i+1}^2 + \frac{1}{2} mg\ell\theta_{i+1}^2 \\ &= \frac{1}{2} m\ell^2 \left(\omega_i - \frac{g}{\ell}\,\theta_i\Delta t\right)^2 + \frac{1}{2} mg\ell\,(\theta_i + \omega_i\Delta t)^2 \\ &= \frac{1}{2} m\ell^2\omega_i^2 + \frac{1}{2} mg\ell\theta_i^2 + \left(\frac{1}{2} m\ell^2\omega_i^2 + \frac{1}{2} mg\ell\theta_i^2\right)\Delta t^2 \\ &= E_i\left(1 + \frac{g}{\ell}\Delta t^2\right). \end{aligned} \tag{2.18}$$

The energy will therefore be a monotonically increasing function of time. While we can make the numerical evolution closer to the expected evolution by reducing the step size Δt, we can never get rid of the fundamental instability of the Euler method of integration: the energy will continue to increase with time no matter how small we make Δt.

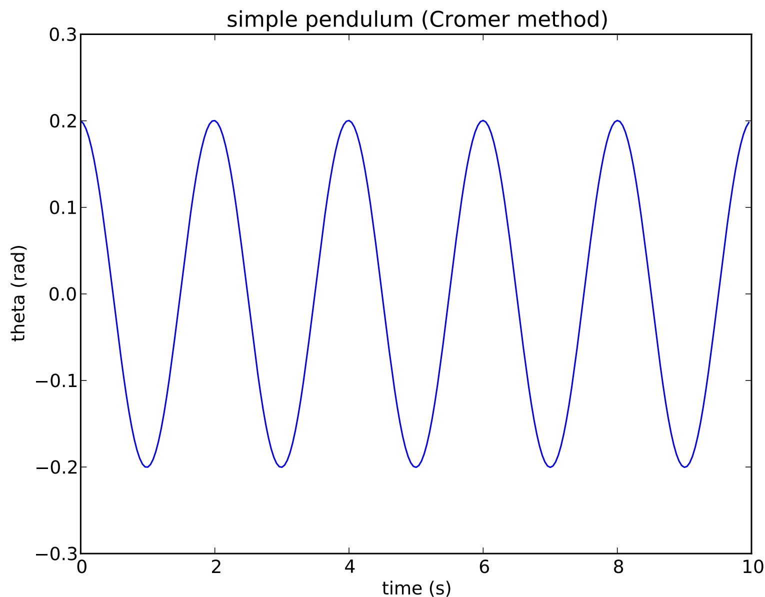

Figure 2.6: Results from running the modified program `pendulum.py` (Listing 2.5 with modification given in Listing 2.6) with the same input as in Fig. (2.5).

Consider the following modification to the numerical scheme:

$$\omega_{i+1} = \omega_i - \frac{g}{\ell}\theta_i \Delta t \tag{2.19a}$$

$$\theta_{i+1} = \theta_i + \omega_{i+1}\Delta t. \tag{2.19b}$$

This is known as the Euler-Cromer method. Note that Eq. (2.19a) is identical with Eq. (2.16a), but in Eq. (2.19b), the value ω_{i+1} is used rather than the value ω_i in Eq. (2.16b). Evaluating the derivative $d\theta/dt$ at the current time step rather than the past time step makes all the difference in this case: the integration scheme is now stable. Consider a modified version of `pendulum.py` with the single change

Listing 2.6: Modification to program `pendulum.py`

```
17      theta[i+1] = theta[i]+omega[i+1]*dt
```

The evolution produced by this modified version of `pendulum.py` with the same input as before is shown in Fig. 2.6.

Exercise 2.4

a) Investigate why the Cromer's modification to the Euler method stabilizes the system. Repeat the calculation of Eq. (2.18) and show that the energy evolves according to

$$E_{i+1} = E_i + \frac{1}{2} mg\ell \left(\omega_i^2 - \frac{g}{\ell}\theta_i^2 \right) \Delta t^2 + O(\Delta t^3). \tag{2.20}$$

Substitute the known solution for $\theta(t)$ and $\omega(t)$ and integrate over one period of oscillation to show that the Euler-Cromer system preserves the energy over one period of oscillation to $O(\Delta t^2)$ while the Euler system only preserves the energy to $O(\Delta t)$. (In fact, the Euler-Cromer system does even better: it preserves the energy to $O(\Delta t^3)$ over each oscillation.)

Modify the program `pendulum.py` to calculate the total energy of the pendulum at each time step and verify explicitly that it is conserved with the Euler-Cromer method. Contrast this to the situation when the Euler method is used.

b) Perform numerical evolutions of the pendulum using the Euler-Cromer method with various step sizes and compare with the analytic solution to compute the truncation error of the integration method. Plot the error as a function of time for various resolutions. Show that the Euler-Cromer method (like the Euler method) is still only a first-order method.

Until now we have only considered ordinary differential for which we already have analytic solutions. Now we'll try something a bit more interesting, and investigate chaos in the double pendulum.

The double pendulum consisting of two bobs of mass m connected with massless rods of length ℓ is shown in Fig. 2.7. The positions of the two bobs are given by

$$x_1 = \ell \sin\theta_1, \qquad y_1 = -\ell \cos\theta_1 \tag{2.21a}$$

$$x_2 = \ell(\sin\theta_1 + \sin\theta_2) \qquad y_2 = -\ell(\cos\theta_1 + \cos\theta_2) \tag{2.21b}$$

and so the kinetic energy and the potential energy are

$$\begin{aligned} T &= \frac{1}{2}m(\dot{x}_1^2 + \dot{y}_1^2 + \dot{x}_2^2 + \dot{y}_2^2) \\ &= \frac{1}{2}m\ell^2[2\dot{\theta}_1^2 + \dot{\theta}_2^2 + 2\dot{\theta}_1\dot{\theta}_2\cos(\theta_1 - \theta_2)] \end{aligned} \tag{2.22}$$

$$\begin{aligned} V &= mgy_1 + mgy_2 \\ &= -mg\ell(2\cos\theta_1 + \cos\theta_2). \end{aligned} \tag{2.23}$$

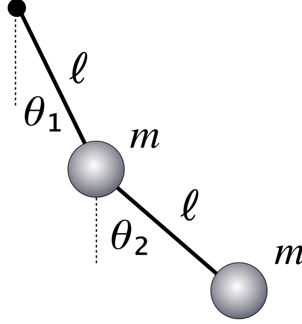

Figure 2.7: The double pendulum.

The Lagrangian for the system is

$$\begin{aligned} L &= T - V \\ &= \frac{1}{2} m\ell^2 [2\dot{\theta}_1^2 + \dot{\theta}_2^2 + 2\dot{\theta}_1 \dot{\theta}_2 \cos(\theta_1 - \theta_2)] + mg\ell(2\cos\theta_1 + \cos\theta_2) \end{aligned} \tag{2.24}$$

from which we obtain the canonical momenta

$$p_{\theta_1} = \frac{\partial L}{\partial \dot{\theta}_1} = \frac{1}{2} m\ell^2 [4\dot{\theta}_1 + 2\dot{\theta}_2 \cos(\theta_1 - \theta_2)] \tag{2.25}$$

$$p_{\theta_2} = \frac{\partial L}{\partial \dot{\theta}_2} = \frac{1}{2} m\ell^2 [2\dot{\theta}_2 + 2\dot{\theta}_1 \cos(\theta_1 - \theta_2)]. \tag{2.26}$$

The Hamiltonian is $H = \dot{\theta}_1 p_{\theta_1} + \dot{\theta}_2 p_{\theta_2} - L$ written in terms of the coordinates and the momenta:

$$H = \frac{1}{2} \frac{p_{\theta_1}^2 + 2p_{\theta_2}^2 - 2p_{\theta_1} p_{\theta_2} \cos(\theta_1 - \theta_2)}{m\ell^2 [1 + \sin^2(\theta_1 - \theta_2)]} - mg\ell(2\cos\theta_1 + \cos\theta_2). \tag{2.27}$$

Now Hamilton's equations of motion are

$$\dot{\theta}_1 = \frac{\partial H}{\partial p_{\theta_1}} = \frac{p_{\theta_1} - p_{\theta_2} \cos(\theta_1 - \theta_2)}{\Delta} \tag{2.28a}$$

$$\dot{\theta}_2 = \frac{\partial H}{\partial p_{\theta_2}} = \frac{2p_{\theta_2} - p_{\theta_1} \cos(\theta_1 - \theta_2)}{\Delta} \tag{2.28b}$$

$$\dot{p}_{\theta_1} = -\frac{\partial H}{\partial \theta_1} = -2mg\ell \sin\theta_1 - A + B \tag{2.28c}$$

$$\dot{p}_{\theta_2} = -\frac{\partial H}{\partial \theta_2} = -mg\ell \sin\theta_2 + A - B \tag{2.28d}$$

where

$$\Delta = m\ell^2[1+\sin^2(\theta_1-\theta_2)] \tag{2.28e}$$

$$A = \frac{p_{\theta_1}p_{\theta_2}\sin(\theta_1-\theta_2)}{\Delta} \tag{2.28f}$$

$$B = \frac{p_{\theta_1}^2+2p_{\theta_2}^2-p_{\theta_1}p_{\theta_2}\cos(\theta_1-\theta_2)}{\Delta^2/m\ell^2}\sin(\theta_1-\theta_2)\cos(\theta_1-\theta_2) \tag{2.28g}$$

Equations (2.28) form a system of first order ordinary differential equations. A discretization of these equations using a Euler-Cromer method is now possible, and a program to evolve the resulting system is `dblpend.py` (note that in this program `q1` and `q2` refer to θ_1 and θ_2, while `p1` and `p2` refer to p_{θ_1}/m and p_{θ_2}/m).

Listing 2.7: Program `dblpend.py`

```
1  import math, pylab
2
3  g = 9.8   # standard freefall (m/s^2)
4  l = input('pendulum length (meters) -> ')
5  theta10 = input('initial angle 1 (radians) -> ')
6  theta20 = input('initial angle 2 (radians) -> ')
7  dt = input('time step (seconds) -> ')
8  tmax = input('time to end of simulation (seconds) -> ')
9  nsteps = int(tmax/dt)
10 t = [0.0]*nsteps
11 p1 = [0.0]*nsteps
12 p2 = [0.0]*nsteps
13 q1 = [0.0]*nsteps
14 q2 = [0.0]*nsteps
15 x1 = [0.0]*nsteps
16 x2 = [0.0]*nsteps
17 y1 = [0.0]*nsteps
18 y2 = [0.0]*nsteps
19
20 # initialize
21 q1[0] = theta10
22 q2[0] = theta20
23 x1[0] = l*math.sin(q1[0])
24 y1[0] = -l*math.cos(q1[0])
25 x2[0] = l*(math.sin(q1[0])+math.sin(q2[0]))
26 y2[0] = -l*(math.cos(q1[0])+math.cos(q2[0]))
27
28 # use Euler-Cromer method to integrate the double pendulum
29 for i in range(nsteps-1):
30     s = math.sin(q1[i]-q2[i])
31     c = math.cos(q1[i]-q2[i])
32     D = l**2*(1+s**2)
33     A = p1[i]*p2[i]*s/D
34     B = (p1[i]**2+2*p2[i]**2-2*p1[i]*p2[i]*c)*s*c*l**2/D**2
```

```
35      p1[i+1] = p1[i]-(2*g*l*math.sin(q1[i])-A+B)*dt
36      p2[i+1] = p2[i]-(g*l*math.sin(q2[i])+A-B)*dt
37      q1[i+1] = q1[i]+(p1[i+1]-p2[i+1]*c)/D*dt
38      q2[i+1] = q2[i]+(2*p2[i+1]-p1[i+1]*c)/D*dt
39      t[i+1] = t[i]+dt
40
41      # put q1 and q2 in range -pi to +pi
42      q1[i+1] = (q1[i+1]+math.pi)%(2.0*math.pi)-math.pi
43      q2[i+1] = (q2[i+1]+math.pi)%(2.0*math.pi)-math.pi
44
45      # also compute (x,y) locations of points 1 and 2
46      x1[i+1] = l*math.sin(q1[i+1])
47      y1[i+1] = -l*math.cos(q1[i+1])
48      x2[i+1] = l*(math.sin(q1[i+1])+math.sin(q2[i+1]))
49      y2[i+1] = -l*(math.cos(q1[i+1])+math.cos(q2[i+1]))
50
51  # plot results
52
53  pylab.figure()
54  pylab.title('double pendulum')
55  pylab.plot(t, q2, label='theta2')
56  pylab.plot(t, q1, label='theta1')
57  pylab.xlabel('t (s)')
58  pylab.ylabel('angle (rad)')
59  pylab.legend(loc=9)
60  pylab.grid()
61  pylab.figure(figsize=(6, 6))
62  pylab.title('Lissajou curves for the double pendulum')
63  pylab.plot(q1, q2)
64  pylab.xlabel('theta1 (rad)')
65  pylab.ylabel('theta2 (rad)')
66  minmax = max(abs(min(q1+q2)), abs(max(q1+q2)))
67  pylab.axis([-minmax, minmax, -minmax, minmax], aspect='equal')
68  pylab.grid()
69  pylab.figure(figsize=(6, 6))
70  pylab.title('double pendulum trace')
71  pylab.plot(x2, y2)
72  pylab.xlabel('x (m)')
73  pylab.ylabel('y (m)')
74  pylab.axis([-2.1, 2.1, -2.1, 2.1], aspect='equal')
75  pylab.grid()
76  pylab.show()
```

For small initial displacement angles, $\theta_1 \ll 1$ and $\theta_2 \ll 1$, the motion is regular as can be seen in Fig. 2.8. However, for larger displacement angles, the motion can become *chaotic* as shown in Fig. 2.9. This means that the evolution of the system depends very sensitively to the initial conditions. In Fig. 2.9 it is seen that two evolutions with very small differences in the initial angle of displacement results in motion that, at first, stay quite near to each other, but then rapidly diverge.

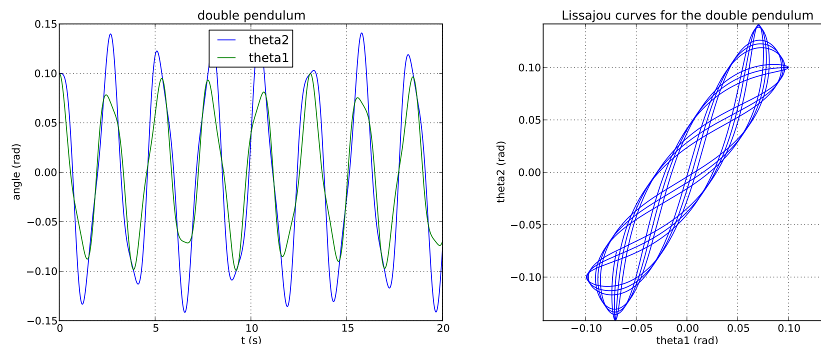

Figure 2.8: Results from running the program `dblpend.py` (Listing 2.7) with input $\ell = 1\,\mathrm{m}$, $\theta_1 = \theta_2 = 0.1\,\mathrm{rad}$, $\Delta t = 0.001\,\mathrm{s}$, and $t_{\mathrm{max}} = 20\,\mathrm{s}$. The left plot shows the evolution of θ_1 and θ_2 with time while the right plot shows the Lissajou curve θ_2 vs. θ_1.

The difficulty with chaotic motion is that, because it is very sensitive to the initial conditions, it is also very sensitive to numerical errors. It is important, therefore, to have highly accurate numerical schemes in order to evolve potentially chaotic systems. We will discuss more accurate numerical schemes in the next section.

Exercise 2.5

a) Investigate the behavior of the double pendulum for small initial displacement angles. Try to find the initial conditions that result in the two normal modes of oscillation. Investigate these normal modes of oscillation.

b) Investigate the chaotic regime. Can you predict where chaotic evolution occurs? How does the step size affect the evolution of a system in for chaotic and non-chaotic motion.

2.4 Orbital motion

The motion of a planets about the Sun under Newtonian gravity is known as the *Kepler problem*. Kepler's three laws can be readily derived for any inverse-squared central force,

$$\boldsymbol{F} = -k\frac{\boldsymbol{r}}{r^3} \tag{2.29}$$

where k is a constant (we leave it arbitrary for the present), $\boldsymbol{r}$ is the separation vector which is directed from the center of force to the body in motion about it, and

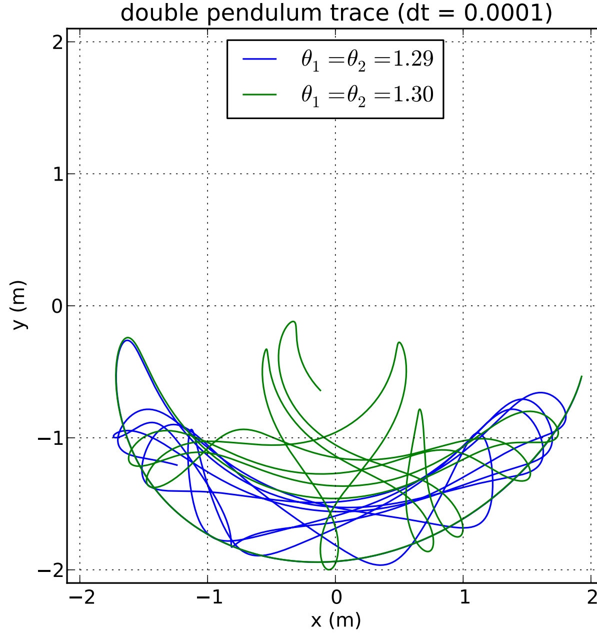

Figure 2.9: Results from running the program `dblpend.py` (Listing 2.7) with input $\ell = 1\,\mathrm{m}$, $\Delta t = 0.0001\,\mathrm{s}$, and $t_{\max} = 12\,\mathrm{s}$ for two slightly different initial angles θ_1 and θ_2. The motion is initially very close but then diverges quite rapidly. This is an example of chaotic motion.

$r = \|\boldsymbol{r}\|$ is the magnitude of this vector. First note that a central force can produce no torque:

$$\boldsymbol{N} = \frac{d\boldsymbol{L}}{dt} = \frac{d}{dt}(\boldsymbol{r} \times \boldsymbol{p}) = \boldsymbol{v} \times \boldsymbol{p} + \boldsymbol{r} \times \boldsymbol{F} = 0 \tag{2.30}$$

where $\boldsymbol{v} = d\boldsymbol{r}/dt$ is the velocity of the body, $\boldsymbol{p} = m\boldsymbol{v}$ is its momentum (m is the mass of the body), and $\boldsymbol{L} = \boldsymbol{r} \times \boldsymbol{p}$ is its angular momentum. The last equality holds since $\boldsymbol{v} \parallel \boldsymbol{p}$ and $\boldsymbol{r} \parallel \boldsymbol{F}$ where $\boldsymbol{F} = d\boldsymbol{p}/dt$, so both cross-products vanish. The conservation of angular momentum implies that $\boldsymbol{r}$ and $\boldsymbol{v}$ span a conserved orbital plane. Furthermore, the area $d\boldsymbol{A}$ swept out in time dt is $\frac{1}{2}\boldsymbol{r} \times d\boldsymbol{r}$ so the *areal velocity*,

$$\frac{d\boldsymbol{A}}{dt} = \frac{1}{2}\boldsymbol{r} \times \boldsymbol{v} = \frac{1}{2m}\boldsymbol{r} \times \boldsymbol{p} = \frac{\boldsymbol{L}}{2m} = \text{const} \tag{2.31}$$

is constant. This is *Kepler's second law*.

To obtain *Kepler's first law*, that the bound orbits are elliptical with the force center at a focus of the ellipse, we begin by considering the evolution of the vector

$\boldsymbol{v} \times \boldsymbol{L}$:

$$\begin{aligned}\frac{d}{dt}(\boldsymbol{v} \times \boldsymbol{L}) &= \frac{d\boldsymbol{v}}{dt} \times \boldsymbol{L} = \boldsymbol{a} \times \boldsymbol{L} \\ &= -\frac{k}{mr^3}[\boldsymbol{r} \times (\boldsymbol{r} \times \boldsymbol{p})] = -\frac{k}{r^3}[\boldsymbol{r} \times (\boldsymbol{r} \times \boldsymbol{v})] \\ &= -\frac{k}{r^3}[(\boldsymbol{r} \cdot \boldsymbol{v})\boldsymbol{r} - (\boldsymbol{r} \cdot \boldsymbol{r})\boldsymbol{v}] \\ &= \frac{d}{dt}\left(k\frac{\boldsymbol{r}}{r}\right)\end{aligned} \tag{2.32}$$

where to obtain the last line we note that $dr/dt = \boldsymbol{v} \cdot \boldsymbol{r}/r$. Since both sides are now total derivatives in time, we have

$$\boldsymbol{v} \times \boldsymbol{L} = k\left(\frac{\boldsymbol{r}}{r} + \boldsymbol{e}\right) \tag{2.33}$$

where $\boldsymbol{e}$ is a constant vector, which is proportional to the *Laplace-Runge-Lenz vector*. We now take the dot product of both sides with the vector $\boldsymbol{r}$ and define the *true anomaly*, θ, by $\cos\theta = \hat{\boldsymbol{r}} \cdot \hat{\boldsymbol{e}}$, to obtain

$$kr(1 + e\cos\theta) = \boldsymbol{r} \cdot (\boldsymbol{v} \times \boldsymbol{L}) = (\boldsymbol{r} \times \boldsymbol{v}) \cdot \boldsymbol{L} = \frac{\boldsymbol{L}}{m} \cdot \boldsymbol{L} = \frac{L^2}{m} = \text{const} \tag{2.34}$$

where $e = \|\boldsymbol{e}\|$ is the *eccentricity* of the orbit, and finally we have

$$r = \frac{L^2/(mk)}{1 + e\cos\theta} = \frac{a(1-e^2)}{1 + e\cos\theta} \tag{2.35}$$

which is the equation of an ellipse of eccentricity e and semi-major axis $a = L^2/[mk(1-e^2)]$ where r is the distance from one of the foci of the ellipse.

The semi-minor axis of an ellipse is $b = a/\sqrt{1-e^2}$ and its area is $A = \pi ab$. This area is swept out in one orbital period, P, and, from Eq. (2.31) we have

$$\frac{L}{2m}P = A = \pi ab = \pi\frac{a^2}{1-e^2} = \pi a^{3/2}\frac{L}{\sqrt{mk}}. \tag{2.36}$$

We then obtain *Kepler's third law*, relating the period of the orbit to its semi-major axis,

$$P^2 = \frac{4\pi^2 m}{k}a^3. \tag{2.37}$$

Although Kepler's first law gives us the *shape* of the orbit (an ellipse), and Kepler's second law describes how much area is swept out in an interval in time, the problem of determining the exact position of the body along the ellipse as a function of time, $\theta(t)$, is more involved. Here we derive *Kepler's equation* for $\theta(t)$. First, the orbit is described in terms of an *eccentric anomaly*, ψ, which is the angle of a fictitious body that moves in a circle that is concentric with the ellipse that the true

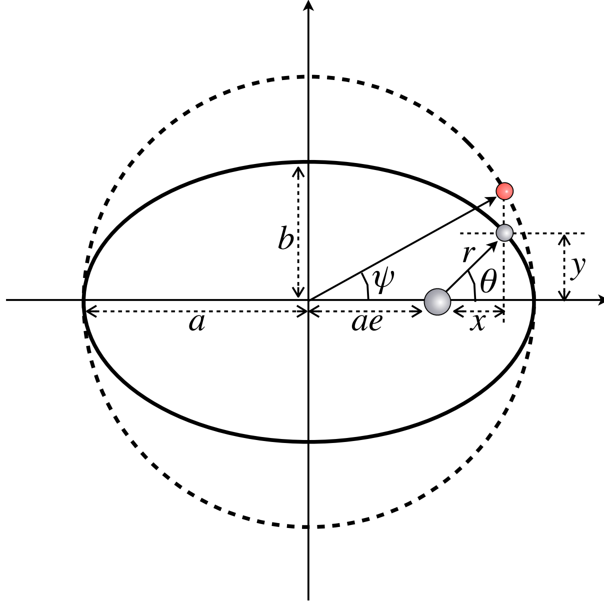

Figure 2.10: The geometric relationship between the eccentric anomaly, ψ, and the true anomaly θ. The fictitious body (red) moves on a circle (thick dotted line) that is concentric to the ellipse (thick solid line) of the true body's orbit so that it always has the same x-coordinate as the true body.

body moves on, where the position of the fictitious body has the same x-coordinate as the true body (see Fig. 2.10). Simple trigonometry gives the relations

$$x = r\cos\theta = a\cos\psi + ae \tag{2.38a}$$

$$y = r\sin\theta = b\sin\psi = a\sqrt{1-e^2}\sin\psi \tag{2.38b}$$

$$r = \frac{a(1-e^2)}{1+e\cos\theta} = a - ae\cos\psi. \tag{2.38c}$$

From these one obtains the relationships

$$r(1+\cos\theta) = a(1-e)(1+\cos\psi) \tag{2.39a}$$

$$r(1-\cos\theta) = a(1+e)(1-\cos\psi) \tag{2.39b}$$

[note: the sum of Eqs. (2.39a) and (2.39b) is twice Eq. (2.38c), the difference is Eq. (2.38a), and the product is the square of Eq. (2.38b)] and the ratio of Eqs. (2.39a) and (2.39b), along with the half angle formula for the tangent, yields a relationship between the eccentric and true anomalies,

$$\tan\frac{\theta}{2} = \sqrt{\frac{1+e}{1-e}}\tan\frac{\psi}{2}. \tag{2.40}$$

Furthermore, the areal velocity is found to be

$$\frac{\pi ab}{P} = \frac{dA}{dt} = \frac{1}{2}r^2\frac{d\theta}{dt} = \frac{1}{2}ab(1-e\cos\psi)\frac{d\psi}{dt}. \tag{2.41}$$

Now we integrate this equation with respect to time to obtain Kepler's equation,

$$M = \psi - e\sin\psi \tag{2.42}$$

where

$$M = \frac{2\pi t}{P} \tag{2.43}$$

is known as the *mean anomaly*.

One more important equation, known as the *vis-viva equation*, is obtained by computing $v^2 = (dx/dt)^2 + (dy/dt)^2$ from the time derivative of Eqs. (2.38a) and (2.38b), along with Eq. (2.41), which gives

$$v^2 = \frac{4\pi^2a^3}{P^2}\left(\frac{2}{r}-\frac{1}{a}\right) = \frac{k}{m}\left(\frac{2}{r}-\frac{1}{a}\right) \tag{2.44}$$

where Kepler's third law is employed to obtain the second equality. Special cases of the vis-viva equation are for the perihelion speed, $v_{\mathrm{perihelion}}$, where $r_{\mathrm{perihelion}} = a(1-e)$, and the aphelion speed, v_{aphelion}, where $r_{\mathrm{aphelion}} = a(1+e)$:

$$v_{\mathrm{perihelion}} = \frac{2\pi a}{P}\sqrt{\frac{1+e}{1-e}} \quad \text{and} \quad v_{\mathrm{aphelion}} = \frac{2\pi a}{P}\sqrt{\frac{1-e}{1+e}}. \tag{2.45}$$

Kepler's method of computing the location of a planet requires us to compute the true anomaly at any given time and then Kepler's first law yields the orbital distance r. To determine the true anomaly at time t, $\theta(t)$, we must (i) compute the mean anomaly M using Eq. (2.43), (ii) solve the transcendental Kepler's equation, Eq. (2.42), for the eccentric anomaly, and (iii) obtain the true anomaly using Eq. (2.40). Step (ii) can be performed efficiently using Newton's method (see Sec. A.2); nevertheless, this approach to determining the position of a planet is somewhat involved, and cannot be simply extended to more complex systems involving perturbations to the system or to the dynamics of more than two bodies. We therefore consider now a numerical approach to planetary motion involving the direct integration of the equations of motion.

Consider the equations of motion for a planet in orbit about the Sun. We know that the orbital plane is conserved, so we wish to evolve Newton's equations of motion in two-dimensions:

$$\frac{d^2x}{dt^2} = -GM_\odot\frac{x}{(x^2+y^2)^{3/2}} \tag{2.46a}$$

$$\frac{d^2y}{dt^2} = -GM_\odot\frac{y}{(x^2+y^2)^{3/2}} \tag{2.46b}$$

where G is Newton's gravitational constant, $M_\odot$ is the mass of the Sun, and their product is known as the *heliocentric gravitational constant*,

$$GM_\odot = 1.327\,124\,420\,99 \times 10^{20}\,\mathrm{m}^3\,\mathrm{s}^{-2}, \tag{2.47}$$

but in this section we will measure distances in astronomical units (AU) (the orbital radius of the Earth) and times in years, so

$$GM_\odot = 4\pi^2\,\mathrm{AU}^3\,\mathrm{yr}^{-2}. \tag{2.48}$$

We express Eqs. (2.46) as a system of first order differential equations:

$$\frac{dx}{dt} = v_x \tag{2.49a}$$
$$\frac{dv_x}{dt} = -GM_\odot \frac{x}{(x^2+y^2)^{3/2}} \tag{2.49b}$$
$$\frac{dy}{dt} = v_y \tag{2.49c}$$
$$\frac{dv_y}{dt} = -GM_\odot \frac{y}{(x^2+y^2)^{3/2}}. \tag{2.49d}$$

As before we simply need to discretize these equations and then perform a numerical integration (with appropriate initial conditions) in order to evolve the orbit of a planet. Let us consider now better methods for evolving ordinary differential equations than the Euler or Euler-Cromer methods.

The most popular integration methods for ordinary differential equations are known as Runge-Kutta methods. Let us consider first the second-order Runge-Kutta method, known as `rk2`. Suppose we wish to solve the ordinary differential equation

$$\frac{dx}{dt} = f(t,x) \tag{2.50}$$

where $f(t,x)$ is some known function. If we have a value $x_i = x(t_i)$ at time t_i, then we wish to compute the value $x_{i+1} = x(t_{i+1})$ at time $t_{i+1} = t_i + \Delta t$, which is

$$\begin{aligned} x(t_i+\Delta t) &= x(t_i) + \left.\frac{dx}{dt}\right|_{t_i,x_i} \Delta t + \frac{1}{2}\left.\frac{d^2x}{dt^2}\right|_{t_i,x_i} \Delta t^2 + O(\Delta t^3) \\ &= x(t_i) + f(t_i,x_i)\Delta t + + \frac{1}{2}\left.\frac{df}{dt}\right|_{t_i,x_i} \Delta t^2 + O(\Delta t^3). \end{aligned} \tag{2.51}$$

Now we have

$$\begin{aligned} \frac{df}{dt} &= \frac{\partial f(x,t)}{\partial t} + \frac{\partial f(x,t)}{\partial x}\frac{dx}{dt} \\ &= \frac{\partial f(x,t)}{\partial t} + \frac{\partial f(x,t)}{\partial x} f(x,t) \end{aligned} \tag{2.52}$$

so we find

$$x_{i+1} = x_i + f(t_i, x_i)\Delta t + \frac{1}{2}\left[\left.\frac{\partial f}{\partial t}\right|_{t_i,x_i} + \left.\frac{\partial f}{\partial x}\right|_{t_i,x_i} f(t_i, x_i) \right] \Delta t^2 + O(\Delta t^3). \tag{2.53}$$

Note that

$$f(t_i + \tfrac{1}{2}\Delta t, x_i + \tfrac{1}{2}\Delta x) \approx f(t_i, x_i) + \left.\frac{\partial f}{\partial t}\right|_{t_i,x_i} \frac{\Delta t}{2} + \left.\frac{\partial f}{\partial x}\right|_{t_i,x_i} \frac{\Delta x}{2} \tag{2.54}$$

to first order in Δt and Δx. Therefore, if we choose $\Delta x = f(t_i, x_i)\Delta t$, we have

$$f(t_i + \tfrac{1}{2}\Delta t, x_i + \tfrac{1}{2}\Delta x) = f(t_i, x_i) + \frac{1}{2}\left[\left.\frac{\partial f}{\partial t}\right|_{t_i,x_i} + \left.\frac{\partial f}{\partial x}\right|_{t_i,x_i} f(t_i, x_i) \right] \Delta t + O(\Delta t^2). \tag{2.55}$$

Comparing this equation to Eq. (2.53) we see

$$x_{i+1} = x_i + f(t_i + \tfrac{1}{2}\Delta t, x_i + \tfrac{1}{2}\Delta x)\,\Delta t + O(\Delta t^3) \tag{2.56}$$

where $\Delta x = f(t_i, x_i)\,\Delta t$. This method therefore has *second order* accuracy (recall the Euler method had only first order accuracy), and is known as the *second order Runge-Kutta method*. In summary, the method, `rk2` is as follows:

$$\Delta x_1 = f(t_i, x_i)\,\Delta t \tag{2.57a}$$

$$\Delta x_2 = f(t_i + \tfrac{1}{2}\Delta t, x_i + \tfrac{1}{2}\Delta x_1)\,\Delta t \tag{2.57b}$$

$$x_{i+1} = x_i + \Delta x_2 + O(\Delta x^3). \tag{2.57c}$$

Equation (2.57a) makes an initial guess as to how much to increment x, and then Eq. (2.57b) refines that guess by evaluating the function $f(t, x)$ at the predicted midpoint.

More commonly used is the fourth-order Runge-Kutta method, `rk4`, which is

$$\Delta x_1 = f(t_i, x_i)\,\Delta t \tag{2.58a}$$

$$\Delta x_2 = f(t_i + \tfrac{1}{2}\Delta t, x_i + \tfrac{1}{2}\Delta x_1)\,\Delta t \tag{2.58b}$$

$$\Delta x_3 = f(t_i + \tfrac{1}{2}\Delta t, x_i + \tfrac{1}{2}\Delta x_2)\,\Delta t \tag{2.58c}$$

$$\Delta x_4 = f(t_i + \Delta t, x_i + \Delta x_3)\,\Delta t \tag{2.58d}$$

$$x_{i+1} = x_i + \tfrac{1}{6}\Delta x_1 + \tfrac{1}{3}\Delta x_2 + \tfrac{1}{3}\Delta x_3 + \tfrac{1}{6}\Delta x_4 + O(\Delta x^5). \tag{2.58e}$$

Exercise 2.6

a) Show algebraically that the system of equations given by Eqs. (2.58) is accurate to fourth order.

b) Write a program to integrate the equations of motion for a simple harmonic oscillator using the Euler-Cromer method, the second-order Runge-Kutta method rk2, and the fourth-order Runge-Kutta method rk4. Compare your evolutions to the known solution for various step sizes and demonstrate that the order of accuracy is what you expect for these methods.

We are now ready to integrate the equations of motion for planetary orbits. We express the system of ordinary differential equations in vector notation

$$\frac{d\boldsymbol{X}}{dt} = \boldsymbol{F}(t,\boldsymbol{X}) \tag{2.59}$$

where the vector $\boldsymbol{X}$ contains the positions and velocities

$$\boldsymbol{X} = \begin{bmatrix} x \\ v_x \\ y \\ v_y \end{bmatrix} \tag{2.60}$$

and the vector $\boldsymbol{F}$ gives their derivatives

$$\boldsymbol{F}(t,\boldsymbol{X}) = \begin{bmatrix} v_x \\ -GM_\odot x/(x^2+y^2)^{3/2} \\ v_y \\ -GM_\odot y/(x^2+y^2)^{3/2} \end{bmatrix}. \tag{2.61}$$

(Note that for this problem the function $\boldsymbol{F}$ does not have any explicit dependence on time.) A program to integrate these equations of motion is `planet.py`.

Listing 2.8: Program `planet.py`

```
import math, pylab

GM = (2.0*math.pi)**2  # heliocentric gravitational constant


# function that implements rk4 integration
def rk4(t, x, f, dt):
    dx1 = f(t, x)*dt
    dx2 = f(t+0.5*dt, x+0.5*dx1)*dt
    dx3 = f(t+0.5*dt, x+0.5*dx2)*dt
    dx4 = f(t+dt, x+dx3)*dt
    return x+dx1/6.0+dx2/3.0+dx3/3.0+dx4/6.0

```

```
14
15  # function that returns dX/dt for the orbital equations of motion
16  def dXdt(t, X):
17      x = X[0]
18      vx = X[1]
19      y = X[2]
20      vy = X[3]
21      r = math.sqrt(x**2+y**2)
22      ax = -GM*x/r**3
23      ay = -GM*y/r**3
24      return pylab.array([vx, ax, vy, ay])
25
26
27  x0 = input('initial x position (au) -> ')
28  y0 = input('initial y position (au) -> ')
29  vx0 = input('initial x velocity (au/yr) -> ')
30  vy0 = input('initial y velocity (au/yr) -> ')
31  dt = input('time step (yr) -> ')
32  tmax = input('time to end of simulation (yr) -> ')
33  nsteps = int(tmax/dt)
34  x = [0.0]*nsteps
35  y = [0.0]*nsteps
36
37  # integrate Newton's equations of motion using rk4;
38  # X is a vector that contains the positions and velocities being integrated
39  X = pylab.array([x0, vx0, y0, vy0])
40  for i in range(nsteps):
41      x[i] = X[0]
42      y[i] = X[2]
43      # update the vector X to the next time step
44      X = rk4(i*dt, X, dXdt, dt)
45  pylab.figure(figsize=(6, 6))
46  pylab.plot(x, y, 'o-')
47  pylab.xlabel('x (au)')
48  pylab.ylabel('y (au)')
49  minmax = 1.1*max(abs(min(x+y)), abs(max(x+y)))
50  pylab.axis([-minmax, minmax, -minmax, minmax], aspect='equal')
51  pylab.grid()
52  pylab.show()
```

The perihelion speed of a planet is

$$v_{\text{perihelion}} = \frac{2\pi a}{P}\sqrt{\frac{1+e}{1-e}} \tag{2.62}$$

where P is the orbital period, a is the semi-major axis of the orbit, and e is the orbital eccentricity. For Mercury these values are $P = 0.241\,\text{yr}$, $a = 0.387\,\text{AU}$, and $e = 0.206$ which yields $v_{\text{perihelion}} = 12.43\,\text{AU}\,\text{yr}^{-1}$ and a perihelion distance $r_{\text{perihelion}} = a(1-e) = 0.307\,\text{AU}$. Running `planet.py` with the input $x_0 = 0.307\,\text{AU}$, $y_0 = 0$,

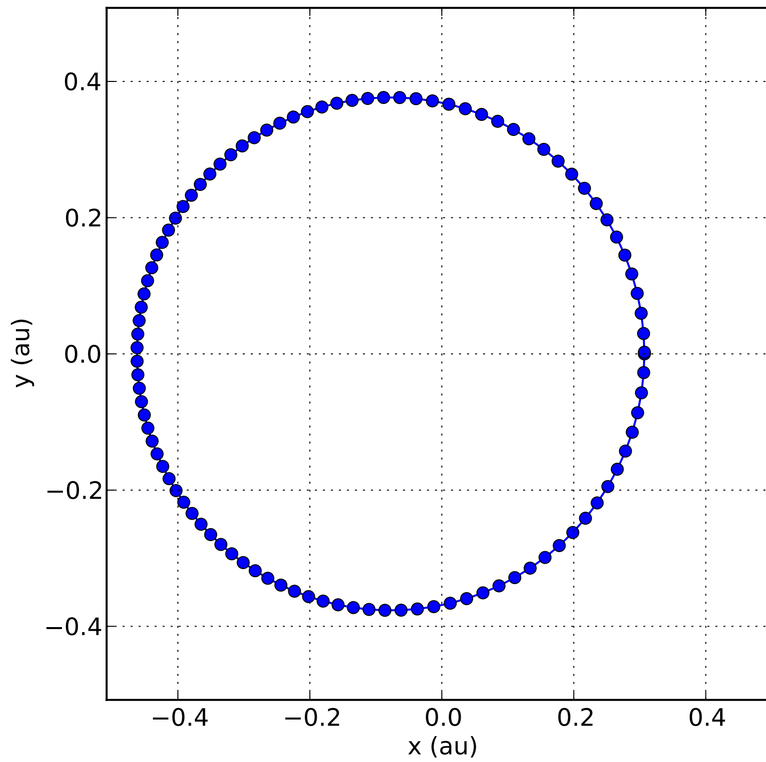

Figure 2.11: Results from running the program `planet.py` (Listing 2.8) with input parameters corresponding to the orbit of Mercury: $x_0 = 0.307\,\mathrm{AU}$, $y_0 = 0$, $v_{x,0} = 0$, $v_{y,0} = 12.43\,\mathrm{AU\,yr^{-1}}$, $\Delta t = 0.00241\,\mathrm{yr}$ and $t_{\mathrm{max}} = 0.241\,\mathrm{yr}$.

$v_{x,0} = 0$, $v_{y,0} = 12.43\,\mathrm{AU\,yr^{-1}}$, $\Delta t = 0.00241\,\mathrm{yr}$ and $t_{\mathrm{max}} = 0.241\,\mathrm{yr}$ produces the results shown in Fig. 2.11.

Exercise 2.7 In General Relativity, orbital motion can be described as in Newtonian gravity but there is an additional $1/r^4$ term in the force law,

$$F(r)/m = \frac{GM_\odot}{r^2}\left(1 + \frac{\alpha}{r^2}\right) \tag{2.63}$$

where

$$\alpha = \frac{3GM_\odot}{c^2}\,a(1-e^2) \tag{2.64}$$

which has the value $\alpha \approx 1.10 \times 10^{-8}\,\mathrm{AU}^2$ for Mercury. Modify the program `planet.py` to include the extra force term due to General Relativity. Also modify the program so that it locates each perihelion (or aphelion) anomaly. For various values of α in the range $0.0001 < \alpha < 0.01$ determine the rate of perihelion advance. Extrapolate this result to the value of α for Mercury.

Suppose we are interested computing the orbit of a comet, such as Halley's comet, having an eccentricity near unity. The orbit of Halley's comet has a semi-major axis $a = 17.8\,\mathrm{AU}$ and an orbital period of 75 yr, which suggests that we should be able to take large time steps, but it also has a eccentricity of $e = 0.967$, so while its aphelion is 35.1 AU, its perihelion is only 0.568 AU, and while it is close to the Sun we will need to take very short time steps. In order to robustly evolve the orbit of Halley's comet we will obtain an *adaptive scheme* for integration of ordinary differential equations where the step size is controlled so that the error is kept near some desired tolerance.

First we need a method for determining how much error is being made with each step. The direct way to determine this is to take the step Δt two ways: once as a single step of size Δt, and a second time as two steps of size $\frac{1}{2}\Delta t$. This difference between the two results then is an estimate of the error being made in the single step. This method is known as *step doubling*.

Alternatively, one could perform the single step using two methods, say once with a fourth-order method and once with a fifth-order method. Again, the difference between the two results is an estimate of the error of the fourth order scheme. It turns out that it is possible to find a fifth-order Runge-Kutta scheme in which the various estimates of Δx can be combined differently in order to form a fourth-order Runge-Kutta scheme. This is useful because one does not need to re-evaluate the function being integrated at different points.

Such a fifth-order Runge-Kutta method with an "embedded" fourth order Runge-Kutta method, `rk45`, uses requires six estimates of Δx:

$$\begin{aligned}
\Delta x_1 &= f(t_i, x_i)\,\Delta t \\
\Delta x_2 &= f(t_i + a_2\Delta t, x_i + b_{21}\Delta x_1)\,\Delta t \\
\Delta x_3 &= f(t_i + a_3\Delta t, x_i + b_{31}\Delta x_1 + b_{32}\Delta x_2)\,\Delta t \\
&\vdots \\
\Delta x_6 &= f(t_i + a_6\Delta t, x_i + b_{61}\Delta x_1 + b_{62}\Delta x_2 + \cdots + b_{65}\Delta x_5)\,\Delta t
\end{aligned} \tag{2.65a}$$

where the as and bs are certain constants given below. Given these values of Δx, a fifth order scheme is

$$\overset{(5)}{x}_{i+1} = x_i + c_1 \Delta x_1 + c_2 \Delta x_2 + \cdots + c_6 \Delta x_6 \tag{2.65b}$$

where the cs are another set of constants, while a fourth order scheme is obtained with a different linear combination of the Δx values:

$$\overset{(4)}{x}_{i+1} = x_i + d_1 \Delta x_1 + d_2 \Delta x_2 + \cdots + d_6 \Delta x_6 \tag{2.65c}$$

where the ds are a different set of constants. Our error estimate is then

$$e = \overset{(5)}{x}_{i+1} - \overset{(4)}{x}_{i+1} = \sum_{j=1}^{6} (c_j - d_j) \Delta x_j. \tag{2.65d}$$

The *Cash-Karp parameters* are given in Eq.(2.66) in Table 2.1.

If we have a desired value for the error, e_0, then an adaptive method refines the step size so as to achieve this value. If the trial step size Δt produces errors that are larger than e_0 then the step size is reduced and the step is repeated. However, error is smaller than e_0 then the current step is accepted, but for the next step the step size is increased. In either case, we can make a prediction of the "correct" step size given the ratio between the estimated error e and the desired error e_0: since the method is fourth-order, the local error is $O(\Delta t^5)$ so the estimated step size to use next is

$$(\Delta t)_{\text{next}} = \Delta t \left| \frac{e_0}{e} \right|^{1/5}. \tag{2.67}$$

Finally, to choose e_0 we must decide whether we want to control the *absolute error* of the result or the *relative error*:

$$e_0 = \epsilon_{\text{abs}} + \epsilon_{\text{rel}} |x|. \tag{2.68}$$

Note that for functions passing through zero, the relative error term can vanish so it is helpful to have some value for ϵ_{abs}.

The program `halley.py` evolves the orbit of Halley's comet using an adaptive Runge-Kutta integrator called `ark45`. The resulting orbit is shown in Fig. 2.12 where we see that the adaptive integrator has a higher concentration of points when the comet is near the Sun and is moving fastest.

Listing 2.9: Program `halley.py`

```
1  import math, pylab
2
3
4  def rk45(t, x, f, dt):
5      """ Fifth order and embedded fourth order Runge-Kutta integration
       step
6      with error estimate. """
7
```

$$a_2 = \frac{1}{5} \quad a_3 = \frac{3}{10} \quad a_4 = \frac{3}{5} \quad a_5 = 1 \quad a_6 = \frac{7}{8} \tag{2.66a}$$

$$b_{21} = \frac{1}{5} \quad b_{31} = \frac{3}{40} \quad b_{41} = \frac{3}{10} \quad b_{51} = -\frac{11}{54} \quad b_{61} = \frac{1631}{55296} \tag{2.66b}$$

$$b_{32} = \frac{9}{40} \quad b_{42} = -\frac{9}{10} \quad b_{52} = \frac{5}{2} \quad b_{62} = \frac{175}{512} \tag{2.66c}$$

$$b_{43} = \frac{6}{5} \quad b_{53} = -\frac{70}{27} \quad b_{63} = \frac{575}{13824} \tag{2.66d}$$

$$b_{54} = \frac{35}{27} \quad b_{64} = \frac{44275}{110592} \tag{2.66e}$$

$$b_{65} = \frac{253}{4096} \tag{2.66f}$$

$$c_1 = \frac{37}{378} \quad c_2 = 0 \quad c_3 = \frac{250}{621} \quad c_4 = \frac{125}{594} \quad c_5 = 0 \quad c_6 = \frac{512}{1771} \tag{2.66g}$$

$$d_1 = \frac{2825}{27648} \quad d_2 = 0 \quad d_3 = \frac{18575}{48384} \quad d_4 = \frac{13525}{55296} \quad d_5 = \frac{277}{14336} \quad d_6 = \frac{1}{4} \tag{2.66h}$$

Table 2.1: Cash-Karp parameters for rk45.

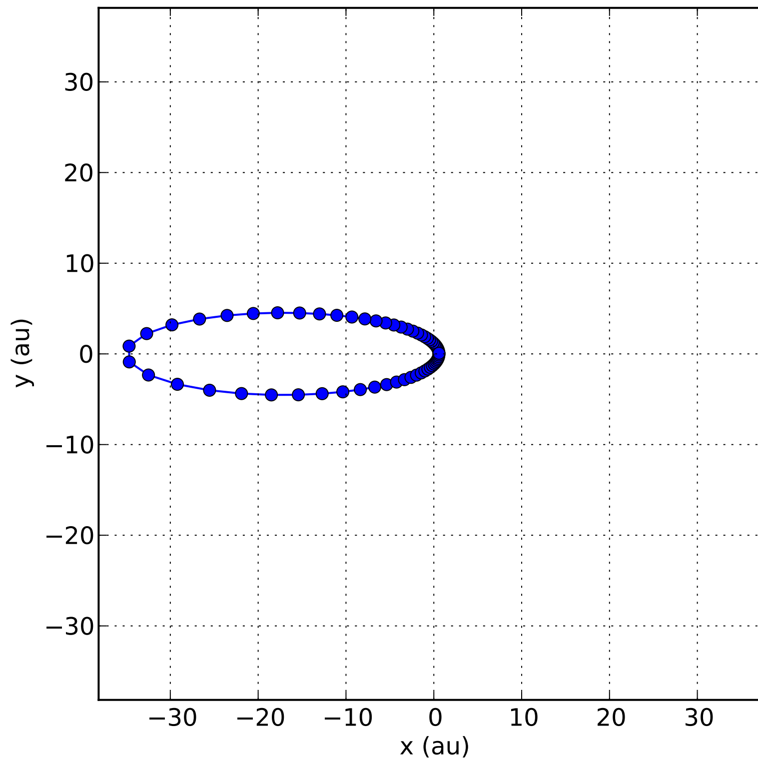

Figure 2.12: Results from running the program `halley.py` (Listing 2.9). Notice that the steps are much closer together near perihelion where the orbital velocity is the greatest.

```
8      # Cash—Karp parameters
9      a2, a3, a4, a5, a6 = 1./5., 3./10., 3./5., 1., 7./8.
10     b21 = 1./5.
11     b31, b32 = 3./40., 9./40.
12     b41, b42, b43 = 3./10., −9./10., 6./5.
13     b51, b52, b53, b54 = −11./54., 5./2., −70./27., 35./27.
14     b61, b62, b63, b64, b65 = 1631./55296., 175./512., 575./13824.,
15                               44275./110592., 253./4096.
16     c1, c2, c3, c4, c5, c6 = 37./378, 0., 250./621., 125./594., 0.,
        512./1771.
17     d1, d2, d3, d4, d5, d6 = 2825./27648., 0., 18575./48384.,
        13525./55296.,
18                              277./14336., 1./4.
19     e1, e2, e3, e4, e5, e6 = c1−d1, c2−d2, c3−d3, c4−d4, c5−d5, c6−d6
```

```
20
21      # evaluate the function at the six points
22      dx1 = f(t, x)*dt
23      dx2 = f(t+a2*dt, x+b21*dx1)*dt
24      dx3 = f(t+a3*dt, x+b31*dx1+b32*dx2)*dt
25      dx4 = f(t+a4*dt, x+b41*dx1+b42*dx2+b43*dx3)*dt
26      dx5 = f(t+a5*dt, x+b51*dx1+b52*dx2+b53*dx3+b54*dx4)*dt
27      dx6 = f(t+a6*dt, x+b61*dx1+b62*dx2+b63*dx3+b64*dx4+b65*dx5)*dt
28      # compute and return the error and the new value of x
29      err = e1*dx1+e2*dx2+e3*dx3+e4*dx4+e5*dx5+e6*dx6
30      return (x+c1*dx1+c2*dx2+c3*dx3+c4*dx4+c5*dx5+c6*dx6, err)
31
32
33  def ark45(t, x, f, dt, epsabs=1e-6, epsrel=1e-6):
34      """ Adaptive Runge-Kutta integration step. """
35
36      safe = 0.9  # safety factor for step estimate
37      # compute the required error
38      e0 = epsabs+epsrel*max(abs(x))
39      dtnext = dt
40      while True:
41          # take a step and estimate the error
42          dt = dtnext
43          (result, error) = rk45(t, x, f, dt)
44          e = max(abs(error))
45          dtnext = dt*safe*(e0/e)**0.2
46          if e < e0:  # accept step: return x, t, and dt for next step
47              return (result, t+dt, dtnext)
48
49
50  GM = (2.*math.pi)**2  # heliocentric gravitational constant
51
52
53  # returns the derivative dX/dt for the orbital equations of motion
54  def dXdt(t, X):
55      x = X[0]
56      vx = X[1]
57      y = X[2]
58      vy = X[3]
59      r = math.sqrt(x**2+y**2)
60      ax = -GM*x/r**3
61      ay = -GM*y/r**3
62      return pylab.array([vx, ax, vy, ay])
63
64
65  # orbital parameters for Halley's comet
66  a = 17.8  # semimajor axis, au
67  e = 0.967  # eccentricity
68  P = a**1.5  # orbital period
69  r1 = a*(1.-e)  # perihelion distance
```

```
70  v1 = (GM*(2/r1-1/a))**0.5   # perihelion speed
71
72  # initial data
73  x0 = r1
74  y0 = 0.
75  vx0 = 0.
76  vy0 = v1
77  dt = 0.01
78  tmax = P
79  x = [x0]
80  y = [y0]
81  t = [0.]
82
83  # integrate Newton's equations of motion using ark45
84  # X is a vector that contains the positions and velocities being integrated
85  X = pylab.array([x0, vx0, y0, vy0])
86  T = 0.
87  while T < tmax:
88      (X, T, dt) = ark45(T, X, dXdt, dt)
89      x += [X[0]]
90      y += [X[2]]
91      t += [T]
92
93  pylab.figure(figsize=(6, 6))
94  pylab.plot(x, y, 'o-')
95  pylab.xlabel('x (au)')
96  pylab.ylabel('y (au)')
97  minmax = 1.1*max(abs(min(x+y)), abs(max(x+y)))
98  pylab.axis([-minmax, minmax, -minmax, minmax], aspect='equal')
99  pylab.grid()
100 pylab.show()
```

Exercise 2.8 Write a program to solve the three-body problem depicted in Fig. 2.13 where the three bodies are initially at rest in the configuration shown and their subsequent motion is determined by Newton's law of gravity. Use units in which $G = 1$ and the masses, distances, and times are dimensionless.

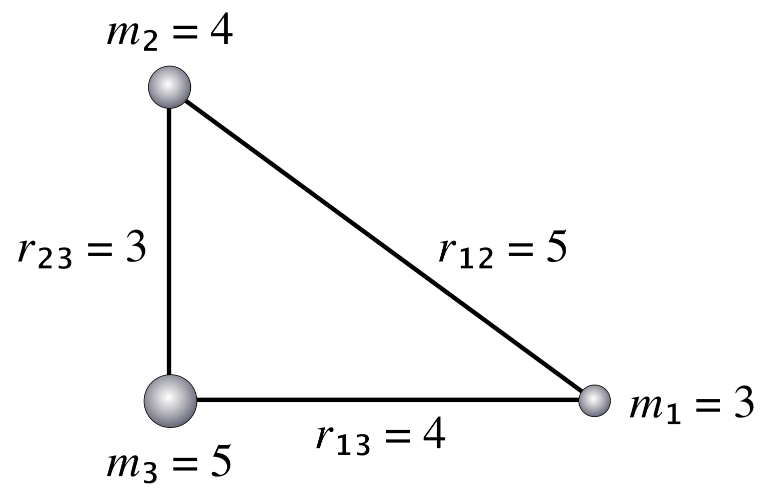

Figure 2.13: Initial configuration for the three body problem.

Chapter 3

Partial differential equations

There are three basic types of partial differential equations. Equations such as the wave equation,

$$\frac{\partial^2 u}{\partial t^2} = c^2 \frac{\partial^2 u}{\partial x^2}, \tag{3.1}$$

where $u(t,x)$ is a displacement function and c is a constant velocity, are known as *hyperbolic equations*. Equations such as the diffusion equation,

$$\frac{\partial u}{\partial t} = \frac{\partial}{\partial x}\left(D \frac{\partial u}{\partial x}\right), \tag{3.2}$$

where $u(t,x)$ is density field and D is the diffusion coefficient are known as *parabolic equations*. The time-dependent Schrödinger equation is another example of a parabolic equation. Equations such as the Poisson equation,

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial x^2} = -\frac{\rho(x,y,z)}{\epsilon_0} \tag{3.3}$$

where $u(x,y,z)$ is a potential function and ρ/ϵ_0 is a source, are known as *elliptic equations*. The time-independent Schrödinger equation is another example of an elliptic equation.

Hyperbolic or parabolic equations are found in *initial value problems*: the field configuration $u(t,x)$ is specified at some initial time and is evolved forward in time. Elliptic equations are found in *boundary value problems*: the value of the field $u(x,y,z)$ is specified on the boundary of a region and we seek the solution through the interior.

3.1 Waves

As our prototype wave equation problem, consider the scenario of propagating a wave pulse along a taught wire of length $L = 1\,\mathrm{m}$ that is fixed at both ends. The

wave equation that we need to solve is

$$\frac{\partial^2 u}{\partial t^2} = c^2 \frac{\partial^2 u}{\partial x^2} \tag{3.4}$$

where $u(t,x)$ is the displacement of the wire and c is the speed of propagation on the wire. For our problem we will assume a value of $c = 300\,\mathrm{m\,s^{-1}}$. In addition to the wave equation we require boundary values and initial conditions. We have already stated that the ends of the wires are fixed, so our boundary values are

$$u(t,0) = u(t,L) = 0. \tag{3.5}$$

As an initial condition we consider a Gaussian pulse

$$u(0,x) = \exp\left(-\frac{(x-x_0)^2}{2\sigma^2}\right) \tag{3.6}$$

of width $\sigma = 0.02\,\mathrm{m}$ centered at position $x_0 = 0.3\,\mathrm{m}$. The wave equation is a *second* order equation so we must specify two initial conditions. As our second initial condition we will assume that the initial displacement has been held fixed up until time $t = 0$ at which point it is released. Our second initial condition is therefore $\partial u(0,x)/\partial t = 0$.

The general solution to the wave equation, Eq. (3.4), is

$$u(t,x) = f(x-ct) + g(x+ct) \tag{3.7}$$

where f and g are arbitrary functions that correspond to a right-moving and a left-moving solution respectively. It is straightforward to substitute this ansatz into the wave equation to verify that it is a solution. The initial conditions are of the form

$$u(0,x) = f(x) + g(x) \qquad \frac{\partial u(0,x)}{\partial t} = c[g'(x) - f'(x)]. \tag{3.8}$$

For example, if we want our initial conditions to correspond to a right-moving wave then we have $f(x) = u(0,x)$ and $g(x) = 0$ while if we want our initial conditions to correspond to a left-moving wave then we have $g(x) = u(0,x)$ and $f(x) = 0$. For our problem we want $\partial u(0,x)/\partial t = 0$ so we find $f(x) = g(x) = \frac{1}{2}u(0,x)$, that is, we have equal parts left-moving and right-moving waves.

Boundary conditions are required at $x = 0$ and $x = L$. For our problem these points are fixed so we simply require

$$u(t,0) = 0 \qquad u(t,L) = 0 \tag{3.9a}$$

$$\frac{\partial u(t,0)}{\partial t} = 0 \qquad \frac{\partial u(t,L)}{\partial t} = 0. \tag{3.9b}$$

These are known as *Dirichlet boundary conditions*. Other types of boundary conditions are possible, e.g., periodic boundary conditions, in which $u(t,0) = u(t,L)$, and outgoing (non-reflecting) boundary condition.

Another method of solving the wave equation algebraically uses the method of *separation of variables*. We assume that the solution $u(t,x)$ can be factored into a function of time alone, $T(t)$, multiplied by a function of position alone, $X(t)$,

$$u(t,x) = T(t)X(x). \tag{3.10}$$

Now we substitute this into the wave equation and we obtain

$$\ddot{T}(t)X(x) = c^2 T(t)X''(x). \tag{3.11}$$

Now divide both sides by $T(t)X(x)$ to obtain

$$\frac{\ddot{T}(t)}{T(t)} = c^2 \frac{X''(x)}{X(x)}. \tag{3.12}$$

Note that the left hand side is a function of time alone and the right hand side is a function of position alone. The only way this can be so is if both the left hand side and the right hand side are constants. Let the constant be ω^2 and let $ck = \omega$. We now have to solve two *ordinary* differential equations

$$\ddot{T}(t) = \omega^2 T(t) \tag{3.13a}$$

$$X''(x) = k^2 X(x). \tag{3.13b}$$

The solutions to these two equations are simply

$$T(t) \propto \mathrm{e}^{\pm i\omega t} \tag{3.14a}$$

$$X''(x) \propto \mathrm{e}^{\pm ikx}. \tag{3.14b}$$

Any value of ω, or k, is possible so the general solution requires us to integrate over all possible values. Our general solution is therefore

$$u(t,x) = \int_{-\infty}^{\infty} dk \left[A(k)\mathrm{e}^{ik(x-ct)} + B(k)\mathrm{e}^{ik(x+ct)}\right]. \tag{3.15}$$

Again we see that our solution is the sum of a right-moving and a left-moving wave. The functions $A(k)$ and $B(k)$ are solved by taking the spatial Fourier transform of the initial data. Imposing the Dirichlet boundary conditions further restrict our solution: the allowed modes must vanish at $x = 0$ and $x = L$ so we have

$$u(t,x) = \sum_{n=1}^{\infty} \{A_n \sin[n\pi(x-ct)/L] + B_n \sin[n\pi(x+ct)/L]\}. \tag{3.16}$$

The coefficients A_n and B_n are determined by the initial data.

To obtain a numerical scheme for solving the wave equation, let us cast it in first-order form with the help of an auxiliary field, $v(t,x)$. We now show that the system of first-order partial differential equations

$$\frac{\partial u(t,x)}{\partial t} = c\frac{\partial v(t,x)}{\partial x} \tag{3.17a}$$

$$\frac{\partial v(t,x)}{\partial t} = c\frac{\partial u(t,x)}{\partial x} \tag{3.17b}$$

is equivalent to the wave equation. First we note that

$$\frac{\partial^2 u(t,x)}{\partial t^2} = c\frac{\partial^2 v(t,x)}{\partial t \partial x} = c^2 \frac{\partial^2 u(t,x)}{\partial x^2} \tag{3.18}$$

where we first used Eq. (3.17a) and then Eq. (3.17b), so if we can define a suitable auxiliary field $v(t,x)$ then the wave equation will hold. Supposing Eq. (3.17b) we see that

$$v(t,x) = c\int_0^t \frac{\partial u(t',x)}{\partial x}\, dt' + f(x) \tag{3.19}$$

where $f(x)$ is an arbitrary function that we can choose so that $c(\partial v/\partial x) = \partial u/\partial t$ at time $t=0$, i.e.,

$$v(0,x) = f(x) = \frac{1}{c}\int^x \frac{\partial u(0,x')}{\partial t}\, dx'. \tag{3.20}$$

Now we simply need to show that $c(\partial v/\partial x) = \partial u/\partial t$ all future times. Since

$$\frac{\partial}{\partial t}\left(c\frac{\partial v}{\partial x} - \frac{\partial u}{\partial t}\right) = c\frac{\partial^2 v}{\partial t \partial x} - \frac{\partial^2 u}{\partial t^2} = c^2\frac{\partial^2 u}{\partial x^2} - \frac{\partial^2 u}{\partial t^2} = 0 \tag{3.21}$$

we conclude that $c(\partial v/\partial x) = \partial u/\partial t$ at all times if it holds initially.

Notice that Eqs. (3.17a) and (3.17b) form a system of equations that are known as *flux-conservative* equations, which have the form

$$\frac{\partial \boldsymbol{u}}{\partial t} + \frac{\partial \boldsymbol{F}(\boldsymbol{u})}{\partial x} = 0 \tag{3.22}$$

where

$$\boldsymbol{u} = \begin{bmatrix} u \\ v \end{bmatrix} \tag{3.23}$$

is the vector of the field variables,

$$\boldsymbol{F}(\boldsymbol{u}) = \begin{bmatrix} 0 & -c \\ -c & 0 \end{bmatrix} \cdot \boldsymbol{u} \tag{3.24}$$

is the matrix of flux terms.

An even simpler example of a flux-conservative equation is the *advection equation*

$$\frac{\partial a(t,x)}{\partial t} + c\frac{\partial a(t,x)}{\partial x} = 0 \tag{3.25}$$

and the solution to such an equation is

$$a(t,x) = f(x - ct) \tag{3.26}$$

for some function f which is determined by the initial conditions $f(x) = a(0,x)$. We see that this initial pulse is “advected” along the x-axis with speed c. We will return to this equation from time to time as a simple analog of the wave equation that we are trying to solve.

For our problem we have initial data for u, and since we are assuming that $\partial u(0,x)/\partial t = 0$, our initial data for the auxiliary function v can be taken to be $v(0,x) = 0$. Finally, to fully specify our problem we need the boundary conditions for u and v. In the case of u the Dirichlet boundary conditions are trivially $u(t,0) = 0$ and $u(t,L) = 0$. This can be achieved by imagining that the string continues into the regions $x < 0$ and $x > L$ and that, at the $x = 0$ boundary, there is always a right-moving wave with $x < 0$ that precisely cancels the left moving wave at that boundary, and similarly, at the $x = L$ boundary, there is a left-moving wave that precisely cancels the right-moving wave. Our boundary conditions for v must ensure that the incoming waves exactly cancel the outgoing waves at the boundary. Our boundary conditions are called *reflective* so that at $x = 0$ only right-moving waves are allowed and at $x = L$ only left-moving waves are allowed. Consider the boundary at $x = 0$. Near this boundary the incoming wave must be $u(t,x) = f(x - ct)$ and so we have

$$\left.\begin{aligned}\frac{\partial u}{\partial x} &= f'(x-ct)\\ \frac{\partial u}{\partial t} &= -cf'(x-ct) = -c\frac{\partial u}{\partial x}\end{aligned}\right\}\quad \text{near } x = 0. \tag{3.27}$$

This means that

$$\frac{\partial v}{\partial x} = \frac{1}{c}\frac{\partial u}{\partial t} = -\frac{\partial u}{\partial x} \qquad \text{near } x = 0. \tag{3.28}$$

That is, the gradient of $u+v$ is zero at the $x = 0$ boundary. Similarly at the boundary $x = L$ only left-moving waves are allowed so the solution must be $u(t,x) = g(x+ct)$ and we find that the gradient of $u - v$ is zero at the $x = L$ boundary.

For our numerical evolution we require a discretized version of our flux-conservative equation and the boundary conditions. Suppose that we divide the length of wire into a grid of points separated by Δx. For spatial derivatives we can use a second-order accurate formula

$$\frac{\partial u(t,x)}{\partial x} = \frac{u(t,x+\Delta x) - u(t,x-\Delta x)}{2\Delta x} + O(\Delta x^2) \tag{3.29}$$

and similarly for $\partial v/\partial x$. For the time derivatives we use Euler differencing,

$$\frac{\partial u(t,x)}{\partial t} = \frac{u(t+\Delta t,x) - u(t,x)}{\Delta t} + O(\Delta t) \tag{3.30}$$

and similarly for $\partial v/\partial t$. This is only first-order accurate in the time step Δt but it has the advantage that we only need to know u at time t to compute u at time $t+\Delta t$. The boundary conditions on v are provided by the requirement that $\partial(u+v)/\partial x = 0$ at $x = 0$ and $\partial(u-v)/\partial x = 0$ at $x = L$, which yield

$$u(t,0) + v(t,0) = u(t,\Delta x) + v(t,\Delta x) \tag{3.31a}$$

$$u(t,L) - v(t,L) = u(t,L-\Delta x) - v(t,L-\Delta x) \tag{3.31b}$$

and since $u(t,0)=0$ and $u(t,L)=0$ we have

$$v(t,0)=v(t,\Delta x)+u(t,\Delta x) \tag{3.32a}$$
$$v(t,L)=v(t,L-\Delta x)-u(t,L-\Delta x). \tag{3.32b}$$

We thus obtain the following scheme for evolving our wave equation: let

$$\begin{aligned}\overset{n}{u}_j &= u(n\,\Delta t, j\,\Delta x)\\ \overset{n}{v}_j &= u(n\,\Delta t, j\,\Delta x)\end{aligned} \tag{3.33}$$

be the fields where $n=0,1,2,\ldots$ are the time steps and $j=0,1,2,\ldots,N$ are the grid points along the wire with $N=L/\Delta x$. Then our system of equations is

$$\overset{n+1}{u}_j = \overset{n}{u}_j + c\,\frac{\overset{n}{v}_{j+1}-\overset{n}{v}_{j-1}}{2\Delta x}\,\Delta t \tag{3.34a}$$
$$\overset{n+1}{v}_j = \overset{n}{v}_j + c\,\frac{\overset{n}{u}_{j+1}-\overset{n}{u}_{j-1}}{2\Delta x}\,\Delta t \tag{3.34b}$$

for $j=1,2,\ldots,N-1$. This scheme is known as a *forward time, centered space* or FTCS scheme. Our boundary conditions are

$$\begin{aligned}&\overset{n}{u}_0 = 0, && \overset{n}{u}_N = 0,\\ &\overset{n}{v}_0 = \overset{n}{v}_1 + \overset{n}{u}_1 && \overset{n}{v}_N = \overset{n}{v}_{N-1} - \overset{n}{u}_{N-1}\,.\end{aligned} \tag{3.35}$$

The program `waveftcs.py` produces an animation of the displacement function $u(t,x)$. The results are shown in Fig. 3.1. We see that the evolution is unstable: numerical inaccuracies are quickly amplified and destroy the evolution in only a few time steps.

Listing 3.1: Program `waveftcs.py`

```
1  import math, pylab
2  
3  # fixed parameters
4  c = 300.0  # speed of propagation, m/s
5  L = 1.0  # length of wire, m
6  x0 = 0.3  # initial pulse location, m
7  s = 0.02  # initial pulse width, m
8  
9  # input parameters
10 dx = L*input('grid spacing in units of wire length (L) -> ')
11 dt = dx/c*input('time step in units of (dx/c) -> ')
12 tmax = L/c*input('evolution time in units of (L/c) -> ')
13 
14 # construct initial data
15 N = int(L/dx)
16 x = [0.0]*(N+1)
```

```
u0 = [0.0]*(N+1)
v0 = [0.0]*(N+1)
u1 = [0.0]*(N+1)
v1 = [0.0]*(N+1)
for j in range(N+1):
    x[j] = j*dx
    u0[j] = math.exp(-0.5*((x[j]-x0)/s)**2)

# prepare animated plot
pylab.ion()
(line, ) = pylab.plot(x, u0, '-k')
pylab.ylim(-1.2, 1.2)
pylab.xlabel('x (m)')
pylab.ylabel('u')

# preform the evolution
t = 0.0
while t < tmax:
    # update plot
    line.set_ydata(u0)
    pylab.title('t = %5f'%t)
    pylab.draw()
    pylab.pause(0.1)

    # derivatives at interior points
    for j in range(1, N):
        v1[j] = v0[j]+0.5*dt*c*(u0[j+1]-u0[j-1])/dx
        u1[j] = u0[j]+0.5*dt*c*(v0[j+1]-v0[j-1])/dx

    # boundary conditions
    u1[0] = u1[N] = 0.0
    v1[0] = v1[1]+u1[1]
    v1[N] = v1[N-1]-u1[N-1]

    # swap old and new lists
    (u0, u1) = (u1, u0)
    (v0, v1) = (v1, v0)
    t += dt

# freeze final plot
pylab.ioff()
pylab.show()
```

To investigate the instability that we will introduce the *von Neumann stability analysis*. First we'll consider the simpler advection equation given by Eq. (3.25). The FTCS system for this equation has the form

$$\overset{n+1}{a}_j = \overset{n}{a}_j - c \, \frac{\overset{n}{a}_{j+1} - \overset{n}{a}_{j-1}}{2\Delta x} \, \Delta t. \tag{3.36}$$

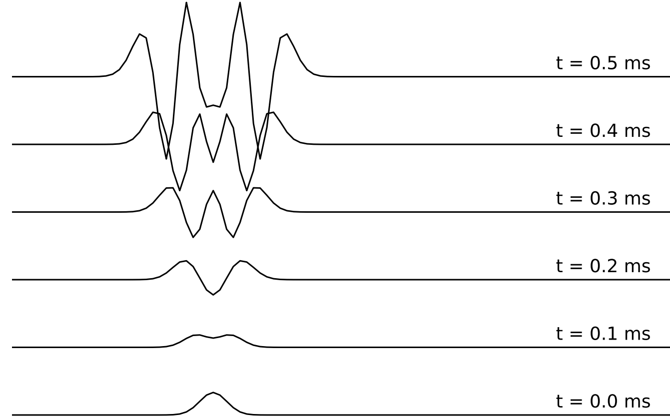

Figure 3.1: Results from running the program `waveftcs.py` (Listing 3.1) with parameters $\Delta x = 0.01$ and $\Delta t = \Delta x/c$. The evolution is unstable.

The eigenmodes are of the form

$$e^{ik(x-ct)} = e^{ikx} e^{-i\omega t} \tag{3.37}$$

where k is a spatial wave number and $kc = \omega$ is the vibration frequency. If we start with a mode of the form

$$\overset{0}{a}_j = e^{ikj\,\Delta x} \tag{3.38}$$

we would expect it to evolve so that at time $t_n = n\,\Delta t$ it would be

$$\overset{n}{a}_j = (e^{-i\omega\,\Delta t})^n e^{ikj\,\Delta x}. \tag{3.39}$$

Let us consider what actually happens to the spatial eigenmodes under the FTCS finite difference approximation to the advection equation. Consider modes of the form

$$\overset{n}{a}_j = \xi^n(k) e^{ikj\,\Delta x} \tag{3.40}$$

where $\xi(k)$ is a complex amplitude for each k. With each time step, the amplitude of the mode changes by one more power of ξ so we see that if $|\xi(k)| > 0$ modes will be exponentially growing while if $|\xi(k)| < 0$ they will be damped. If we substitute Eq. (3.40) into Eq. (3.36) we obtain

$$\xi(k) = 1 - ic\frac{\Delta t}{\Delta x}\sin k\,\Delta x \tag{3.41}$$

so

$$|\xi(k)|^2 = 1 + \left(c\frac{\Delta t}{\Delta x}\right)^2 \sin^2 k\,\Delta x. \tag{3.42}$$

This shows that *all* modes are unstable under the FTCS differencing scheme, and the modes with $k \sim 1/\Delta x$ will grow the fastest.

A simple cure to the instability of the FTCS method is to make the replacement

$$\overset{n}{a}_j \rightarrow \frac{1}{2}(\overset{n}{a}_{j+1} + \overset{n}{a}_{j-1})$$

in the finite difference equations; the resulting scheme is known as the *Lax method*,

$$\overset{n+1}{a}_j = \frac{1}{2}(\overset{n}{a}_{j+1} + \overset{n}{a}_{j-1}) - c\,\frac{\overset{n}{a}_{j+1} - \overset{n}{a}_{j-1}}{2\Delta x}\,\Delta t. \tag{3.43}$$

Repeating the von Neumann stability analysis shows that with the Lax scheme we have

$$\xi(k) = \cos k\,\Delta x - ic\frac{\Delta t}{\Delta x}\sin k\,\Delta x \tag{3.44}$$

or

$$|\xi(k)|^2 = 1 + \left[\left(c\frac{\Delta t}{\Delta x}\right)^2 - 1\right]\sin^2 k\,\Delta x. \tag{3.45}$$

The evolution is stable if $|\xi| \le 1$, which translates to the requirement

$$c\,\Delta t \le \Delta x. \tag{3.46}$$

This requirement is known as the *Courant condition*. It ensures that all of the points that could causally affect a point being computed are considered.

Applying the Lax method to the wave equation problem we obtain the finite differencing scheme

$$\overset{n+1}{u}_j = \frac{1}{2}(\overset{n}{u}_{j+1} + \overset{n}{u}_{j-1}) + c\,\frac{\overset{n}{v}_{j+1} - \overset{n}{v}_{j-1}}{2\Delta x}\,\Delta t \tag{3.47a}$$

$$\overset{n+1}{v}_j = \frac{1}{2}(\overset{n}{v}_{j+1} + \overset{n}{v}_{j-1}) + c\,\frac{\overset{n}{u}_{j+1} - \overset{n}{u}_{j-1}}{2\Delta x}\,\Delta t. \tag{3.47b}$$

The only changes we need to make to the program `waveftcs.py` is in the finite differencing equations. Here we list the lines in which our new program `wavelax.py` differs from `waveftcs.py`.

Listing 3.2: Modified lines in program `wavelax.py`

```
43          v1[j] = 0.5*(v0[j-1]+v0[j+1])+0.5*dt*c*(u0[j+1]-u0[j-1])/dx
44          u1[j] = 0.5*(u0[j-1]+u0[j+1])+0.5*dt*c*(v0[j+1]-v0[j-1])/dx
```

In Fig. 3.2 we see that the evolution is quite stable when $c\Delta t = \Delta x$ but in Fig. 3.3 we encounter instability when $c\Delta t > \Delta x$ as expected. In Fig. 3.4 we see

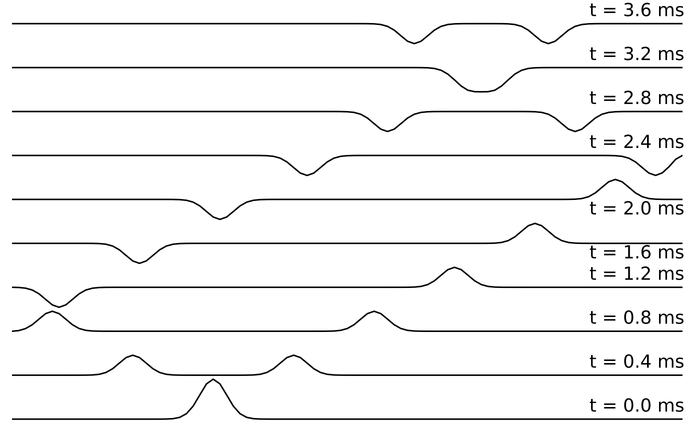

Figure 3.2: Results from running the program `wavelax.py` (Listing 3.2) with parameters $\Delta x = 0.01$ and $\Delta t = \Delta x/c$. The evolution is stable.

that the Lax equations are stable when $c\Delta t < \Delta x$, but also that there is numerical dissipation of the wave.

We can repeat the von Neumann stability analysis for the two-variable system we use to solve the wave equation. We assume that the eigenmodes are of the form

$$\begin{bmatrix} \overset{n}{u}_j \\ \overset{n}{v}_j \end{bmatrix} = \xi^n(k)\mathrm{e}^{ikj\Delta x} \begin{bmatrix} \overset{0}{u} \\ \overset{0}{v} \end{bmatrix} \tag{3.48}$$

where $\overset{0}{u}$ and $\overset{0}{v}$ are constants. If we put this ansatz into the FTCS differencing scheme of Eq. (3.34) we obtain the equation

$$\begin{bmatrix} 1-\xi(k) & ic\dfrac{\Delta t}{\Delta x}\sin k\Delta x \\ ic\dfrac{\Delta t}{\Delta x}\sin k\Delta x & 1-\xi(k) \end{bmatrix} \cdot \begin{bmatrix} \overset{0}{u} \\ \overset{0}{v} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}. \tag{3.49}$$

This is an eigenvalue equation for the eigenvalues $\xi(k)$ and the only values that admit solutions are the values for which the determinant of the matrix is zero. The roots of the characteristic equation are

$$\xi(k) = 1 \pm ic\frac{\Delta t}{\Delta x}\sin k\Delta x. \tag{3.50}$$

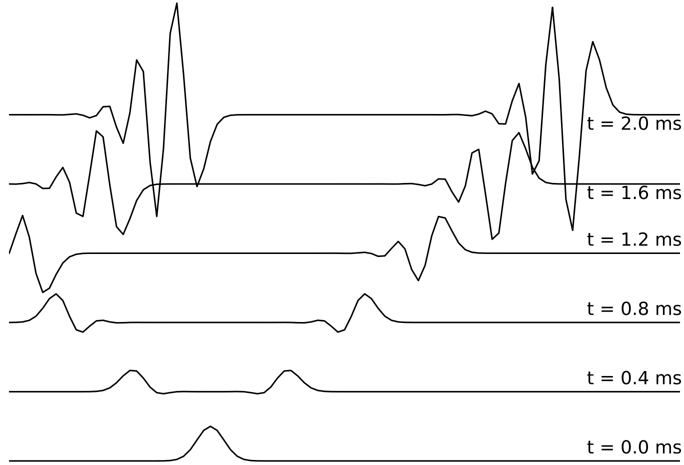

Figure 3.3: Results from running the program `wavelax.py` (Listing 3.2) with parameters $\Delta x = 0.01$ and $\Delta t = 1.1\Delta x/c$. The Courant condition is violated and the evolution is unstable.

Again we see that the FTCS scheme is unconditionally unstable.

For the Lax differencing scheme, however, the eigenvalue equation takes the form

$$\begin{bmatrix} \cos k\Delta x - \xi(k) & ic\dfrac{\Delta t}{\Delta x}\sin k\Delta x \\ ic\dfrac{\Delta t}{\Delta x}\sin k\Delta x & \cos k\Delta x - \xi(k) \end{bmatrix} \cdot \begin{bmatrix} \overset{0}{u} \\ \overset{0}{v} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}. \tag{3.51}$$

and the roots of the characteristic equation are

$$\xi(k) = \cos k\Delta x \pm ic\frac{\Delta t}{\Delta x}\sin k\Delta x. \tag{3.52}$$

As expected, we find that $|\xi(k)| > 1$ when $c\,\Delta t > \Delta x$, which indicates unstable evolution, $|\xi(k)| = 1$ when $c\,\Delta t = \Delta x$, which is the Courant condition and indicates stable evolution with no amplitude dissipation, and $|\xi(k)| < 1$ when $c\,\Delta t < \Delta x$, which again indicates stable evolution but now with amplitude dissipation.

Recall that we have adopted differencing schemes that were second order accurate in space but only first order accurate in time. Let us now consider a method that is second-order accurate in time.

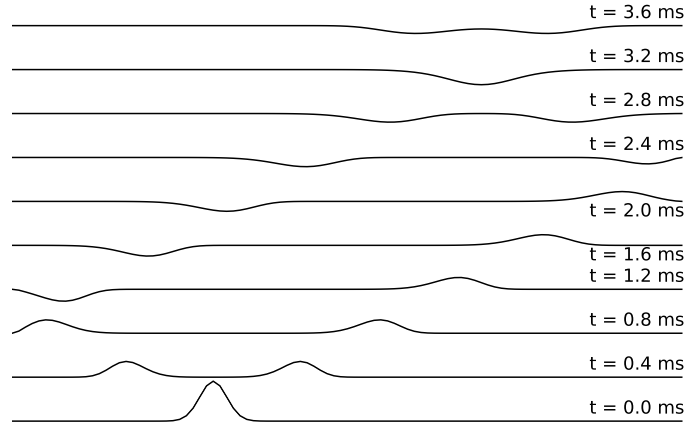

Figure 3.4: Results from running the program `wavelax.py` (Listing 3.2) with parameters $\Delta x = 0.01$ and $\Delta t = 0.9\Delta x/c$. The evolution is stable but there is numerical dissipation of the wave.

The method we will consider is called a *leapfrog method* because we evolve one of our two fields, the u field, on a lattice that has locations at integer multiples of Δx and Δt, while we evolve the other field, the v field, on a lattice that is offset by $\frac{1}{2}\Delta x$ and $\frac{1}{2}\Delta t$. The two fields then play leapfrog with each time step going between the points that were used to compute the gradients. See Fig. 3.5.

Our discretization is therefore as follows:

$$\overset{n}{u}_j = u(n\,\Delta t, j\,\Delta x) \qquad 0 \le j \le N \tag{3.53a}$$

$$\overset{n}{v}_j = v(n\,\Delta t - \tfrac{1}{2}\Delta t, j\,\Delta x + \tfrac{1}{2}\Delta x) \qquad 0 \le j \le N-1. \tag{3.53b}$$

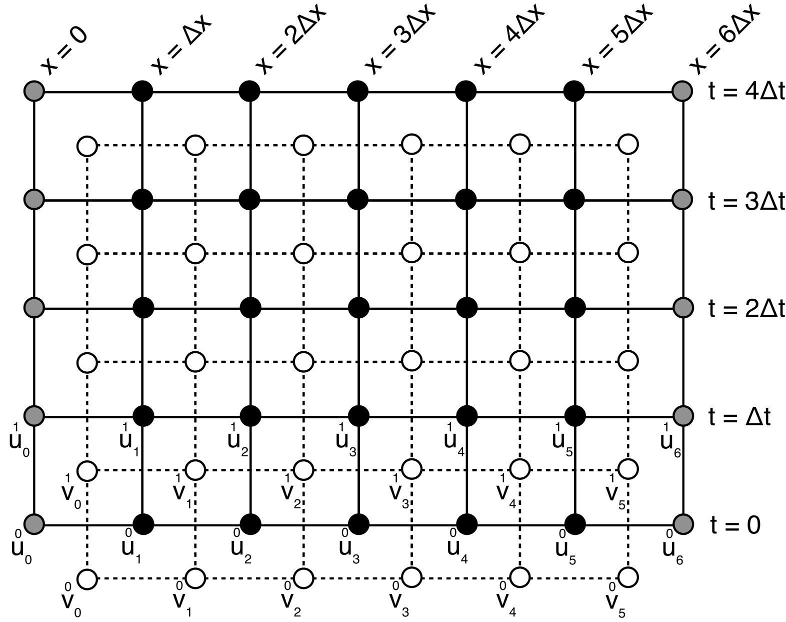

Figure 3.5: The offset grid meshes used in the leapfrog method. The filled circles represent the points where the u field is evaluated — at integer steps in Δx and Δt — while the open circles represent the points where the v field is evaluated. The circles that are filled with grey are boundary points that must be supplied by the boundary conditions. Notice that there are no boundary points for the v field.

Using the Taylor expansion of u about the point $(t+\frac{1}{2}\Delta t, x)$ we have

$$u(t+\Delta t,x) = u(t+\tfrac{1}{2}\Delta t,x) + (\tfrac{1}{2}\Delta t)\frac{\partial u(t+\frac{1}{2}\Delta t,x)}{\partial t} + \frac{1}{2}(\tfrac{1}{2}\Delta t)^2\frac{\partial^2 u(t+\frac{1}{2}\Delta t,x)}{\partial t^2} + O(\Delta t^3) \tag{3.54a}$$

$$u(t,x) = u(t+\tfrac{1}{2}\Delta t,x) - (\tfrac{1}{2}\Delta t)\frac{\partial u(t+\frac{1}{2}\Delta t,x)}{\partial t} + \frac{1}{2}(\tfrac{1}{2}\Delta t)^2\frac{\partial^2 u(t+\frac{1}{2}\Delta t,x)}{\partial t^2} + O(\Delta t^3) \tag{3.54b}$$

and so by subtracting the second equation from the first we have

$$\frac{\partial u(t+\frac{1}{2}\Delta t,x)}{\partial t} = \frac{u(t+\Delta t,x) - u(t,x)}{\Delta t} + O(\Delta t^2). \tag{3.55}$$

Similarly we perform a Taylor expansion of v about the same point,

$$\begin{aligned} v(t+\tfrac{1}{2}\Delta t, x+\tfrac{1}{2}\Delta x) &= v(t+\tfrac{1}{2}\Delta t, x) + (\tfrac{1}{2}\Delta x)\frac{\partial v(t+\frac{1}{2}\Delta t, x)}{\partial x} \\ &\quad + \frac{1}{2}(\tfrac{1}{2}\Delta x)^2 \frac{\partial^2 v(t+\frac{1}{2}\Delta t, x)}{\partial x^2} + O(\Delta x^3) \qquad (3.56a) \\ v(t+\tfrac{1}{2}\Delta t, x-\tfrac{1}{2}\Delta x) &= v(t+\tfrac{1}{2}\Delta t, x) - (\tfrac{1}{2}\Delta x)\frac{\partial v(t+\frac{1}{2}\Delta t, x)}{\partial x} \\ &\quad + \frac{1}{2}(\tfrac{1}{2}\Delta x)^2 \frac{\partial^2 v(t+\frac{1}{2}\Delta t, x)}{\partial x^2} + O(\Delta x^3) \qquad (3.56b) \end{aligned}$$

so

$$\frac{\partial v(t+\frac{1}{2}\Delta t, x)}{\partial x} = \frac{v(t+\frac{1}{2}\Delta t, x+\frac{1}{2}\Delta x) - v(t+\frac{1}{2}\Delta t, x-\frac{1}{2}\Delta x)}{\Delta x} + O(\Delta x^2). \quad (3.57)$$

Since $\partial u/\partial t = c(\partial v/\partial x)$ at this point we have

$$\frac{u(t+\Delta t, x) - u(t, x)}{\Delta t} = c\frac{v(t+\frac{1}{2}\Delta t, x+\frac{1}{2}\Delta x) - v(t+\frac{1}{2}\Delta t, x-\frac{1}{2}\Delta x)}{\Delta x} \quad (3.58)$$

up to second order in Δt and Δx. The equivalent formula in terms of our discretization is

$$\frac{\overset{n+1}{u}_j - \overset{n}{u}_j}{\Delta t} = c\frac{\overset{n+1}{v}_j - \overset{n+1}{v}_{j-1}}{\Delta x}. \quad (3.59)$$

Now by evaluating $\partial v/\partial t$ and $\partial u/\partial x$ at the point $(t, x+\frac{1}{2}\Delta x)$ and using the relation $\partial v/\partial t = c\partial u/\partial x$ at this point we arrive at the relation

$$\frac{v(t+\frac{1}{2}\Delta t, x+\frac{1}{2}\Delta x) - v(t-\frac{1}{2}\Delta t, x+\frac{1}{2}\Delta x)}{\Delta t} = c\frac{u(t, x+\Delta x) - u(t, x)}{\Delta x} \quad (3.60)$$

or

$$\frac{\overset{n+1}{v}_j - \overset{n}{v}_j}{\Delta t} = c\frac{\overset{n}{u}_{j+1} - \overset{n}{u}_j}{\Delta x}. \quad (3.61)$$

Therefore, the system of equations are evolved in the leapfrog scheme by the equations

$$\overset{n+1}{v}_j = \overset{n}{v}_j + c\frac{\overset{n}{u}_{j+1} - \overset{n}{u}_j}{\Delta x}\Delta t \quad (3.62a)$$

$$\overset{n+1}{u}_j = \overset{n}{u}_j + c\frac{\overset{n+1}{v}_j - \overset{n+1}{v}_{j-1}}{\Delta x}\Delta t. \quad (3.62b)$$

For our problem the boundary conditions are very simple: we continue to require

$$\overset{n}{u}_0 = 0 \qquad \overset{n}{u}_N = 0 \quad (3.63)$$

but now we do not need to supply boundary conditions at all for v!

Notice that if we substitute Eq. (3.62a) into Eq. (3.62b) then we obtain the equation

$$\frac{\overset{n+1}{u}_j - 2\overset{n}{u}_j + \overset{n-1}{u}_j}{(\Delta t)^2} = c^2 \frac{\overset{n}{u}_{j+1} - 2\overset{n}{u}_j + \overset{n}{u}_{j-1}}{(\Delta x)^2} \tag{3.64}$$

which is just the result we would obtain with a simple finite differencing of the original second-order wave equation, Eq. (3.4). We could have solved the wave equation directly from this form, but in order to compute the field at step $n+1$ we would need the values of the field both at step n and at step $n-1$. The leapfrog method is entirely equivalent but instead of keeping the data for u at two previous time steps, it keeps the data for u and for v.

A complete listing of the program `leapfrog.py` is given below.

Listing 3.3: Program `leapfrog.py`

```
1  import math, pylab
2
3  # fixed parameters
4  c = 300.0  # speed of propagation, m/s
5  L = 1.0  # length of wire, m
6  x0 = 0.3  # initial pulse location, m
7  s = 0.02  # initial pulse width, m
8
9  # input parameters
10 dx = L*input('grid spacing in units of wire length (L) -> ')
11 dt = dx/c*input('time step in units of (dx/c) -> ')
12 tmax = L/c*input('evolution time in units of (L/c) -> ')
13
14 # construct initial data
15 N = int(L/dx)
16 x = [0.0]*(N+1)
17 u0 = [0.0]*(N+1)
18 v0 = [0.0]*N
19 u1 = [0.0]*(N+1)
20 v1 = [0.0]*N
21 for j in range(N+1):
22     x[j] = j*dx
23     u0[j] = math.exp(-0.5*((x[j]-x0)/s)**2)
24
25 # prepare animated plot
26 pylab.ion()
27 (line, ) = pylab.plot(x, u0, '-k')
28 pylab.ylim(-1.2, 1.2)
29 pylab.xlabel('x (m)')
30 pylab.ylabel('u')
31
32 # preform the evolution
33 t = 0.0
```

```
while t < tmax:
    # update plot
    line.set_ydata(u0)
    pylab.title('t = %5f'%t)
    pylab.draw()
    pylab.pause(0.1)

    # derivatives at interior points
    for j in range(N):
        v1[j] = v0[j]+dt*c*(u0[j+1]-u0[j])/dx
    for j in range(1, N):
        u1[j] = u0[j]+dt*c*(v1[j]-v1[j-1])/dx

    # boundary conditions for u
    u1[0] = u1[N] = 0.0

    # swap old and new lists
    (u0, u1) = (u1, u0)
    (v0, v1) = (v1, v0)
    t += dt

# freeze final plot
pylab.ioff()
pylab.show()
```

Figures 3.6 and 3.7 show evolutions obtained from the `leapfrog.py` program using $c\,\Delta t = \Delta x$ and $c\,\Delta t = \frac{1}{2}\Delta x$ respectively. In particular, notice the evolution depicted in Fig. 3.7 has far less dispersion than was seen in the Lax method. A von Neumann stability analysis indicates why. If we substitute our standard ansatz

$$\overset{n}{u}_j = \xi^n(k) \mathrm{e}^{ikj\,\Delta x}\ \overset{0}{u} \tag{3.65}$$

into Eq. (3.64) we find

$$\xi(k) - 2[1 - \alpha^2(1 - \cos k\Delta x)]\xi(k) + 1 = 0 \tag{3.66}$$

where $\alpha = c\Delta t/\Delta x$, which has roots

$$\xi(k) = b \pm \sqrt{b^2 - 1} \tag{3.67}$$

with

$$b = 1 - \alpha^2(1 - \cos k\Delta x). \tag{3.68}$$

If $b^2 \leq 1$ then $\xi = b \pm i\sqrt{1 - b^2}$ and so $|\xi(k)| = 1$ for all k and the evolution is stable. However, if $b^2 > 1$ then for one of the solutions, $|\xi(k)| > 1$ and the evolution is unstable. Therefore, the stability requirement is $|b| \leq 1$ or

$$|1 - \alpha^2(1 - \cos k\Delta x)| \leq 1. \tag{3.69}$$

The right hand side is largest when $\cos k\Delta x = -1$ which produces the inequality $2\alpha^2 - 1 \leq 1$ and so we see again that the condition for stability is the Courant condition, $\alpha \leq 1$.

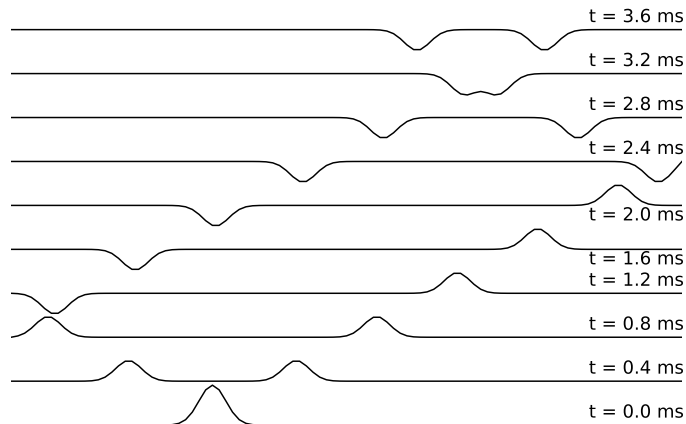

Figure 3.6: Results from running the program `leapfrog.py` (Listing 3.3) with parameters $\Delta x = 0.01$ and $\Delta t = \Delta x / c$. The evolution is stable.

Exercise 3.1

a) Investigate what happens to reflections off of the boundary if you use Neumann boundary conditions, in which the end points are allowed move freely, rather than Dirichlet boundary conditions. Neumann boundary conditions can be achieved as

$$\overset{n}{u}_0 = \overset{n}{u}_1 \quad \text{and} \quad \overset{n}{u}_N = \overset{n}{u}_{N-1} \, .$$

b) Investigate what happens if the initial data is zero but the left-hand boundary is constrained to move in a sinusoid

$$\overset{n}{u}_0 = A \sin \omega t_n$$

for various values of A and ω.

c) Consider a string that is composed of two segments in which the speed is $c_1 = 300\,\mathrm{m\,s^{-1}}$ for $0 \leq x \leq \frac{1}{2}L$ and $c_2 = 150\,\mathrm{m\,s^{-1}}$ for $\frac{1}{2}L < x \leq L$. Starting with a pulse in the left-hand side, study what happens when the pulse hits the boundary between the two components of the string. Repeat the investigation with $c_2 = 600\,\mathrm{m\,s^{-1}}$.

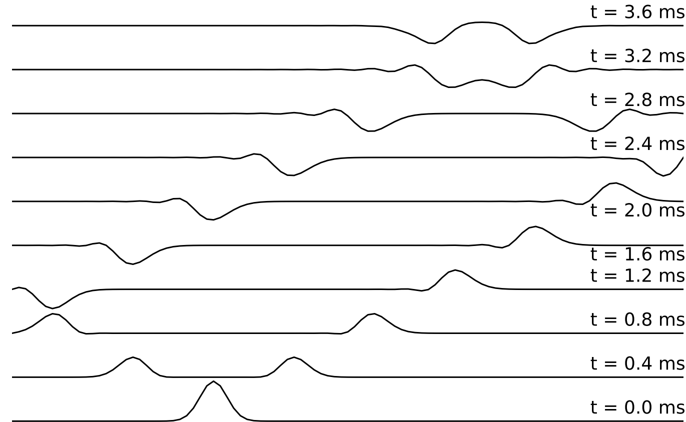

Figure 3.7: Results from running the program `leapfrog.py` (Listing 3.3) with parameters $\Delta x = 0.01$ and $\Delta t = 0.5\Delta x/c$. The evolution is stable. Although there are some inaccuracies in the evolution, there is not the level of dispersion seen in Fig. 3.4.

3.2 Heat diffusion

Let us now consider an initial value problem described by a parabolic partial differential equation.

Suppose a uniform rod of length $L = 1\,\mathrm{m}$ that is insulated along its length but not at its ends. Initially the rod has a temperature $0\,^\circ\mathrm{C}$. At $t = 0$ its left end at position $x = 0$ is put into thermal contact with a heat reservoir of temperature $T_0 = 25\,^\circ\mathrm{C}$ while the right end at position $x = L$ is kept at temperature $0\,^\circ\mathrm{C}$. Let $u(t, x)$ be the temperature along the rod as a function of time. This temperature distribution then satisfies the heat diffusion equation

$$\frac{\partial u}{\partial t} = D\frac{\partial^2 u}{\partial x^2} \tag{3.70}$$

where D is the *thermal diffusivity*,

$$D = \frac{k}{c_p \rho} \tag{3.71}$$

with k being the thermal conductivity, c_p the specific heat capacity at constant pressure, and ρ the mass density. For a copper rod, $D \approx 10^{-4}\,\mathrm{m^2\,s^{-1}}$.

Before we attempt to solve this problem numerically, we will obtain the solution algebraically. In addition to Eq. (3.70) we have the initial condition

$$u(0,x) = 0 \tag{3.72}$$

and the boundary conditions

$$\begin{aligned} u(t,0) &= T_0 \\ u(t,L) &= 0 \end{aligned} \qquad t > 0. \tag{3.73}$$

Let us write our solution as the sum of a particular solution and a homogeneous solution:

$$u(t,x) = u_\mathrm{P}(t,x) + u_\mathrm{H}(t,x). \tag{3.74}$$

A particular solution to Eq. (3.70) with boundary conditions given by Eq. (3.73) is simply

$$u_\mathrm{P}(t,x) = T_0\left(1 - \frac{x}{L}\right) \tag{3.75}$$

which is a stationary solution (it doesn't depend on time), but of course it does not satisfy our initial conditions given by Eq. (3.72). Hence we add a homogeneous solution chosen to Eq. (3.70) that satisfies the boundary conditions

$$\begin{aligned} u_\mathrm{H}(t,0) &= u(t,0) - u_\mathrm{P}(t,0) = 0 \\ u_\mathrm{H}(t,L) &= u(t,L) - u_\mathrm{P}(t,L) = 0 \end{aligned} \qquad t > 0 \tag{3.76}$$

and then require that it solve the initial condition

$$u_\mathrm{H}(0,x) = u(0,x) - u_\mathrm{P}(0,x) = T_0\left(\frac{x}{L} - 1\right). \tag{3.77}$$

Again we will use separation of variables, so we write

$$u_\mathrm{H}(t,x) = T(t)X(x) \tag{3.78}$$

and we find that our partial differential equation separates into the ordinary differential equations

$$\frac{\dot{T}}{T} = D\frac{X''}{X} = -\frac{1}{\tau} \tag{3.79}$$

where $-1/\tau$ is our separation constant. The general solution for $X(x)$ that satisfies the boundary conditions Eq. (3.76) is

$$X(x) = \sum_{n=1}^{\infty} c_n \sin\frac{n\pi x}{L} \tag{3.80}$$

where c_n are coefficients and the separation constants are $\tau_n = L^2/(n^2\pi^2 D)$. The solution for $T(t)$ for each τ_n is simply

$$T(t) = \mathrm{e}^{-t/\tau_n} \tag{3.81}$$

so our homogeneous solution is

$$u_{\mathrm{H}}(t,x)=\sum_{n=1}^{\infty}c_n e^{-t/\tau_n}\sin\frac{n\pi x}{L}. \tag{3.82}$$

We use our initial condition, Eq. (3.77), to determine the coefficients c_n. At $t=0$ we have

$$\sum_{n=1}^{\infty}c_n\sin\frac{n\pi x}{L}=T_0\left(\frac{x}{L}-1\right). \tag{3.83}$$

Multiply both sides by $\sin(m\pi x/L)$ and integrate over x from 0 to L to find

$$\begin{aligned}\frac{L}{2}c_m &= T_0\int_0^L\left(\frac{x}{L}-1\right)\sin\frac{m\pi x}{L}\,dx\\ &= -\frac{T_0}{m\pi}L.\end{aligned} \tag{3.84}$$

Therefore our full solution is

$$u(t,x)=T_0\left\{1-\frac{x}{L}-2\sum_{n=1}^{\infty}\frac{\sin(n\pi x/L)}{n\pi}e^{-t/\tau_n}\right\} \tag{3.85a}$$

with

$$\tau_n=\frac{L^2}{\pi^2 D}\frac{1}{n^2}. \tag{3.85b}$$

At late times, the $n=1$ mode will be dominant and this solution is useful; however, it is not so useful at early times when a large number of terms in the series are important.

Let us now consider a numerical solution to the problem. As before we will begin with a first-order accurate differencing of $\partial u/\partial t$ a second-order-accurate differencing of $\partial^2 u/\partial t^2$. We find

$$\frac{u(t+\Delta t,x)-u(t,x)}{\Delta t}=D\frac{u(t,x+\Delta x)-2u(t,x)+u(t,x-\Delta x)}{(\Delta x)^2}. \tag{3.86}$$

If we discretize our field so that

$$\overset{n}{u}_j=u(n\,\Delta t,j\,\Delta x) \tag{3.87}$$

for $n=0,1,2,\ldots$ and $j=0,1,2,\ldots,N$ where $N=L/\Delta x$, then we have the FTCS evolution scheme

$$\overset{n+1}{u}_j=\overset{n}{u}_j+D\frac{\overset{n}{u}_{j+1}-2\overset{n}{u}_j+\overset{n}{u}_{j-1}}{(\Delta x)^2}\Delta t \tag{3.88}$$

for $n=0,1,2,\ldots$ and $j=1,2,\ldots,N-1$ along with the initial conditions

$$\overset{0}{u}_j=0 \tag{3.89}$$

and the boundary conditions

$$\overset{n}{u}_0 = T_0 \qquad \text{and} \qquad \overset{n}{u}_N = 0. \tag{3.90}$$

Since the FTCS scheme was unstable when applied to the wave equation, it is important to check the stability of this scheme when applied to the diffusion equation. As before we will perform a von Neumann stability analysis by taking the ansatz

$$\overset{n}{u}_j = \xi^n(k) e^{ikj\,\Delta x} \; \overset{0}{u} \tag{3.91}$$

and insert it into Eq. (3.88). This time we find

$$\xi(k) = 1 - 4\frac{D\,\Delta t}{(\Delta x)^2}\sin^2\left(\frac{k\,\Delta x}{2}\right) \tag{3.92}$$

so provided

$$\Delta t \leq \frac{1}{2}\frac{(\Delta x)^2}{D} \tag{3.93}$$

we have $|\xi(k)| \leq 1$ for all k and the evolution will be stable.

The program `heatftcs.py` implements the FTCS scheme.

Listing 3.4: Program `heatftcs.py`

```
import math, pylab

# fixed parameters
T0 = 25.0  # temperature gradient, C
D = 1e-4  # thermal diffusivity, m^2/s
L = 1.0  # length of rod, m

# input parameters
dx = L*input('grid spacing in units of rod length (L) -> ')
dt = dx**2/D*input('time step in units of (dx^2/D) -> ')
tmax = L**2/D*input('evolution time in units of (L^2/D) -> ')

# construct initial data
N = int(L/dx)
x = [0.0]*(N+1)
u0 = [0.0]*(N+1)
u1 = [0.0]*(N+1)
for j in range(N+1):
    x[j] = j*dx

# prepare animated plot
pylab.ion()
(line, ) = pylab.plot(x, u0, '-k')
pylab.ylim(0, T0)
pylab.xlabel('x (m)')
pylab.ylabel('Temperature (Celcius)')
```

```
27
28 # preform the evolution
29 t = 0.0
30 while t < tmax:
31     # update plot
32     line.set_ydata(u0)
33     pylab.title('t = %5f'%t)
34     pylab.draw()
35     pylab.pause(0.1)
36
37     # derivatives at interior points
38     for j in range(1, N):
39         u1[j] = u0[j]+dt*D*(u0[j+1]-2.0*u0[j]+u0[j-1])/dx**2
40
41     # boundary conditions
42     u1[0] = T0
43     u1[N] = 0.0
44
45     # swap old and new lists
46     (u0, u1) = (u1, u0)
47     t += dt
48
49 # freeze final plot
50 pylab.ioff()
51 pylab.show()
```

You can try running this program but you will find it is *painfully* slow. The reason is that the step size is limited to $\Delta t \sim (\Delta x)^2/D$ but the diffusion time scale across the rod takes place on time scales of $\tau \sim L^2/D$. This means that we will need to evolve the system for $\sim (\tau/\Delta t) \sim (L/\Delta x)^2$ time steps. Since $\Delta x \ll L$, this will be a very large number of steps. We want to find a evolution scheme that allows us to take much larger time steps.

The approach we will take is called a *implicit differencing scheme*. Again we perform a first-order-accurate finite difference approximation to $\partial u/\partial t$ and a second-order-accurate finite difference approximation $\partial^2 u/\partial x^2$, but now we evaluate the spatial derivative *at the future time step*. We now have

$$\frac{\overset{n+1}{u}_j - \overset{n}{u}_j}{\Delta t} = D\frac{\overset{n+1}{u}_{j+1} - 2\overset{n+1}{u}_j + \overset{n+1}{u}_{j-1}}{(\Delta x)^2}. \tag{3.94}$$

If we substitute the ansatz of Eq. (3.91) into this equation and solve for ξ we find

$$\xi(k) = \frac{1}{1 + 4\dfrac{D\,\Delta t}{(\Delta x)^2}\sin^2\left(\dfrac{k\,\Delta x}{2}\right)}. \tag{3.95}$$

The amplification factor is *always* less than unity, $|\xi(k)| < 1$ for all k and all step sizes Δt, so the method is *unconditionally* stable. For very large time steps, $\Delta t \to$

∞, we see that Eq. (3.94) is driven towards the form $\partial^2 u/\partial x^2 = 0$, for which the solution is the known late-time particular solution $u_{\rm P}$.

However, if the time steps are too large then the details of the evolution will be lost. For large scale features, those of length scale $\lambda \gg \Delta x$ where $\lambda = 2\pi/k$, then $|\xi| \approx 1$ provided $\Delta t \ll \lambda^2/D$, i.e., we need to take steps that are small compared to the diffusion time over the length scales of interest in order to have an accurate evolution.

Now we turn to the question of how to perform the evolution. Equation (3.94) requires us to evaluate the spatial derivative at the *future* time step, but this is what we are trying to calculate! We can write Eq. (3.94) in the form

$$\alpha \overset{n+1}{u}_{j-1} + \beta \overset{n+1}{u}_{j} + \gamma \overset{n+1}{u}_{j+1} = \overset{n}{u}_{j} \qquad \text{for } j = 1, 2, 3, \ldots, N-1 \tag{3.96}$$

where

$$\alpha = \gamma = -\frac{D\,\Delta t}{(\Delta x)^2} \quad \text{and} \quad \beta = 1 + 2\frac{D\,\Delta t}{(\Delta x)^2}. \tag{3.97}$$

We can express this linear algebra problem in matrix form as

$$\begin{bmatrix} \beta & \gamma & 0 & 0 & 0 & 0 & \cdots & 0 \\ \alpha & \beta & \gamma & 0 & 0 & 0 & \cdots & 0 \\ 0 & \alpha & \beta & \gamma & 0 & 0 & \cdots & 0 \\ 0 & 0 & \alpha & \beta & \gamma & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 & \alpha & \beta & \gamma & 0 \\ 0 & 0 & \cdots & 0 & 0 & \alpha & \beta & \gamma \\ 0 & 0 & \cdots & 0 & 0 & 0 & \alpha & \beta \end{bmatrix} \cdot \begin{bmatrix} \overset{n+1}{u}_1 \\ \overset{n+1}{u}_2 \\ \overset{n+1}{u}_3 \\ \overset{n+1}{u}_4 \\ \vdots \\ \overset{n+1}{u}_{N-3} \\ \overset{n+1}{u}_{N-2} \\ \overset{n+1}{u}_{N-1} \end{bmatrix} = \begin{bmatrix} \overset{n}{u}_1 - \alpha \overset{n+1}{u}_0 \\ \overset{n}{u}_2 \\ \overset{n}{u}_3 \\ \overset{n}{u}_4 \\ \vdots \\ \overset{n}{u}_{N-3} \\ \overset{n}{u}_{N-2} \\ \overset{n}{u}_{N-1} - \gamma \overset{n+1}{u}_N \end{bmatrix} \tag{3.98}$$

or, more succinctly, as

$$\mathbf{A} \cdot \boldsymbol{u} = \boldsymbol{v} \tag{3.99}$$

and note that $\overset{n+1}{u}_0$ and $\overset{n+1}{u}_N$ which appear in the vector $\boldsymbol{v}$ are provided by the boundary conditions.

Methods to solve linear algebra problems such as this one are described in Appendix A.1. In particular, the matrix in Eq. (3.98) is known as a *tridiagonal matrix* since it has non-zero values only along the diagonal, the sub-diagonal, and the super-diagonal. In such cases there are very efficient methods for solving the linear problem. Here we simply present the method to solve the system of equations; Appendix (A.1) provides more detail.

Our method performs two passes over j, one forward and the other backward. First we set the components of the vector $\boldsymbol{v}$ given the current values of the field u

and the boundary conditions:

$$v_j = \begin{cases} \overset{n}{u}_1 - \alpha \overset{n+1}{u}_0 & j = 1 \\ \overset{n}{u}_j & j = 2, 3, \ldots, N-2 \\ \overset{n}{u}_{N-1} - \gamma \overset{n+1}{u}_N & j = N-1. \end{cases} \tag{3.100}$$

Then we perform the following forward sweep:

$$\begin{aligned} u_1 &= v_1/\beta \\ v_1 &= \gamma/\beta \end{aligned} \qquad \text{for } j = 1 \tag{3.101}$$

$$\begin{aligned} u_j &= \frac{v_j - \alpha u_{j-1}}{\beta - \alpha v_{j-1}} \\ v_j &= \frac{\gamma}{\beta - \alpha v_{j-1}} \end{aligned} \qquad \text{for } j = 2, 3, \ldots, N-1. \tag{3.102}$$

Finally we perform the backward sweep to get all the components of $\boldsymbol{u}$:

$$u_j = u_j - u_{j+1} v_j \qquad \text{for } j = N-2, N-3, \ldots 1. \tag{3.103}$$

That's it! At the end of this procedure we have the values for u at time step $n+1$. The vector $\boldsymbol{v}$ is altered in this procedure, but we don't care about that.

This implicit scheme is implemented in the program `implicit.py`.

Listing 3.5: Program `implicit.py`

```
import math, pylab

# fixed parameters
T0 = 25.0  # temperature gradient, C
D = 1e-4  # thermal diffusivity, m^2/s
L = 1.0  # length of rod, m

# input parameters
dx = L*input('grid spacing in units of rod length (L) -> ')
dt = (L**2/D)*input('time step in units of (L^2/D) -> ')
tmax = (L**2/D)*input('evolution time in units of (L^2/D) -> ')

# coefficients of the tridiagonal matrix
alpha = gamma = -D*dt/dx**2
beta = 1.0+2.0*D*dt/dx**2

# construct initial data
N = int(L/dx)
x = [0.0]*(N+1)
v = [0.0]*(N+1)
u = [0.0]*(N+1)
for j in range(N+1):
    x[j] = j*dx
```

```
24 u[0] = T0
25
26 # prepare animated plot
27 pylab.ion()
28 (line, ) = pylab.plot(x, u, '-k')
29 pylab.ylim(0, T0)
30 pylab.xlabel('x (m)')
31 pylab.ylabel('Temperature (Celcius)')
32
33 # preform the evolution
34 t = 0.0
35 while t < tmax:
36     # update plot
37     line.set_ydata(u)
38     pylab.title('t = %5f'%t)
39     pylab.draw()
40     pylab.pause(0.1)
41
42     # swap u and v
43     (u, v) = (v, u)
44
45     # boundary conditions
46     u[0] = T0
47     u[N] = 0.0
48
49     # set the j=1 and j=N-1 points of v to the correct values
50     v[1] -= alpha*u[0]
51     v[N-1] -= gamma*u[N]
52
53     # forward sweep
54     u[1] = v[1]/beta
55     v[1] = gamma/beta
56     for j in range(2, N):
57         den = beta-alpha*v[j-1]
58         u[j] = (v[j]-alpha*u[j-1])/den
59         v[j] = gamma/den
60     # backward sweep
61     for j in reversed(range(1, N-1)):
62         u[j] -= u[j+1]*v[j]
63     t += dt
64
65 pylab.ioff()
66 pylab.show()
```

The results of this code can be seen in Fig. 3.8.

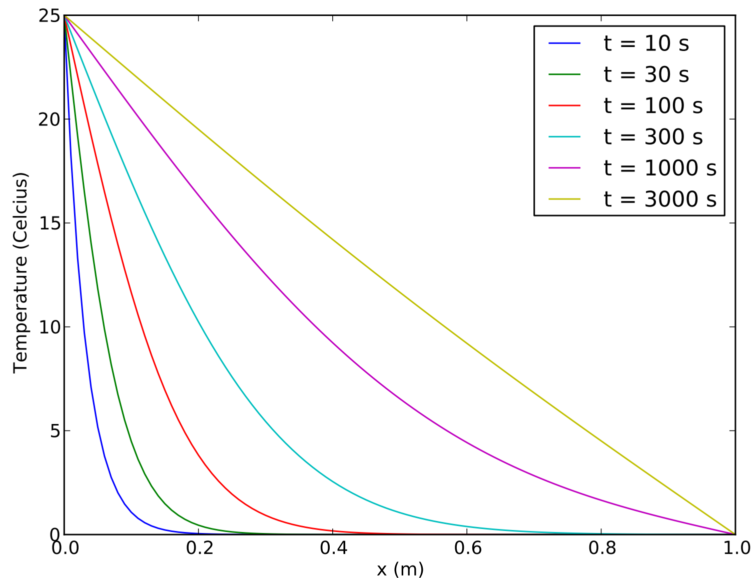

Figure 3.8: Results from running the program `implicit.py` (Listing 3.5) with parameters $\Delta x = 0.01\,L$ and $\Delta t = 0.001\,L^2/D$.

Exercise 3.2

a) Obtain an exact solution to the diffusion equation

$$\frac{\partial u}{\partial t} = D\frac{\partial^2 u}{\partial x^2}$$

on an infinite line (i.e., $-\infty < x < \infty$) with initial condition

$$u(0, x) = \frac{1}{\sqrt{2\pi\sigma^2}}\mathrm{e}^{-\frac{1}{2}x^2/\sigma^2}$$

by assuming the solution retains the same form but with time-dependent σ. Obtain an equation describing how σ evolves in time.

b) Perform a numerical simulation of the diffusion equation with this initial data to confirm your exact solution.

3.3 Schrödinger equation

The Schrödinger wave equation in one dimension is

$$i\hbar\frac{\partial\Psi}{\partial t} = -\frac{\hbar^2}{2m}\frac{\partial^2\Psi}{\partial x^2} + V(x)\Psi \tag{3.104}$$

where $V(x)$ is the potential, $\Psi(t,x)$ is the wave function, and

$$\hbar = 6.626068\times 10^{-34}\,\mathrm{m}^2\,\mathrm{kg}\,\mathrm{s}^{-1} \tag{3.105}$$

is Planck's constant. This is another example of a parabolic partial differential equation.

The solution to Schrödinger's equation is

$$\Psi(t,x) = \mathrm{e}^{-i\hat{H}t/\hbar}\,\Psi(0,x) \tag{3.106}$$

where

$$\hat{H} = -\frac{\hbar^2}{2m}\frac{\partial^2}{\partial x^2} + V(x) \tag{3.107}$$

is the Hamiltonian operator. Notice that the time evolution operator, $\exp(i\hat{H}t/\hbar)$, is a *unitary* operator, so it will preserve the requirement that $|\Psi(t,x)|^2$ represents a probability density:

$$\int_{-\infty}^{\infty} |\Psi(t,x)|^2\,dx = 1. \tag{3.108}$$

We would like our numerical evolution to also preserve this.

The trick to achieving this is to use *Cayley's form* for the finite difference representation of the unitary time evolution operator:

$$\mathrm{e}^{-i\hat{H}\,\Delta t/\hbar} \simeq \frac{1-\frac{1}{2}i\hat{H}\,\Delta t/\hbar}{1+\frac{1}{2}i\hat{H}\,\Delta t/\hbar} \tag{3.109}$$

which is second-order accurate in time. If we let

$$\overset{n}{\Psi}_j = \Psi(n\,\Delta t, j\,\Delta x) \tag{3.110}$$

then Eq. (3.106) and Eq. (3.109) give us

$$(1+\tfrac{1}{2}i\hat{H}\,\Delta t/\hbar)\overset{n+1}{\Psi}_j = (1-\tfrac{1}{2}i\hat{H}\,\Delta t/\hbar)\,\overset{n}{\Psi}_j \tag{3.111}$$

where the operator $\hat{H}$ has the action

$$\hat{H}\,\overset{n}{\Psi}_j = -\frac{\hbar^2}{2m}\frac{\overset{n}{\Psi}_{j+1} - 2\,\overset{n}{\Psi}_j + \overset{n}{\Psi}_{j-1}}{(\Delta x^2)} + V_j\,\overset{n}{\Psi}_j \tag{3.112}$$

where $V_j = V(j\,\Delta x)$. Note that a second-order accurate form of the spatial derivative $\partial^2\Psi/\partial x^2$ has been taken. This method is known as the *Crank-Nicolson method*.

Let us first perform a von Neumann stability analysis. Using the now familiar ansatz

$$\overset{n}{\Psi}_j = \xi^n(k)\mathrm{e}^{ikj\,\Delta x}\,\overset{0}{\Psi} \tag{3.113}$$

we see from Eq. (3.111) that

$$\xi(k) = \frac{1 - i\frac{1}{2}\Omega_j\,\Delta t}{1 + i\frac{1}{2}\Omega_j\,\Delta t} \tag{3.114a}$$

where

$$\Omega_j = \frac{\hbar}{m}\frac{1-\cos k\Delta x}{(\Delta x)^2} + \frac{V_j}{\hbar} \tag{3.114b}$$

and clearly $|\xi(k)| = 1$ for all k and Δt. The Crank-Nicolson method therefore is stable and preserves unitarity for any step size.

As with the implicit method described in the previous section, the Crank-Nicolson equations can be written in the form of a linear algebra problem with a tridiagonal matrix. We have

$$\alpha\,\overset{n+1}{\Psi}_{j-1} + \beta_j\,\overset{n+1}{\Psi}_j + \gamma\,\overset{n+1}{\Psi}_{j+1} = -\alpha\,\overset{n}{\Psi}_{j-1} + (2-\beta_j)\,\overset{n}{\Psi}_j - \gamma\,\overset{n}{\Psi}_{j+1} \tag{3.115a}$$

where the coefficients α, β_i, and γ are

$$\alpha = \gamma = -i\frac{\hbar}{4m}\frac{\Delta t}{(\Delta x)^2} \tag{3.115b}$$

$$\beta_j = 1 + i\frac{\hbar}{2m}\frac{\Delta t}{(\Delta x)^2} + i\frac{V_j}{\hbar}\,\Delta t \tag{3.115c}$$

and so the system of equations we need to solve is

$$\mathbf{A}\cdot\boldsymbol{x} = \boldsymbol{b} \tag{3.116a}$$

with

$$\boldsymbol{x} = \begin{bmatrix} \overset{n+1}{\Psi}_1 \\ \overset{n+1}{\Psi}_2 \\ \overset{n+1}{\Psi}_3 \\ \vdots \\ \overset{n+1}{\Psi}_{N-3} \\ \overset{n+1}{\Psi}_{N-2} \\ \overset{n+1}{\Psi}_{N-1} \end{bmatrix} \tag{3.116b}$$

$$\boldsymbol{b} = \begin{bmatrix} -\alpha \overset{n}{\Psi}_0 + (2-\beta_1)\overset{n}{\Psi}_1 - \gamma \overset{n}{\Psi}_2 - \alpha \overset{n+1}{\Psi}_0 \\ -\alpha \overset{n}{\Psi}_1 + (2-\beta_2)\overset{n}{\Psi}_2 - \gamma \overset{n}{\Psi}_3 \\ -\alpha \overset{n}{\Psi}_2 + (2-\beta_3)\overset{n}{\Psi}_3 - \gamma \overset{n}{\Psi}_4 \\ \vdots \\ -\alpha \overset{n}{\Psi}_{N-4} + (2-\beta_{N-3})\overset{n}{\Psi}_{N-3} - \gamma \overset{n}{\Psi}_{N-2} \\ -\alpha \overset{n}{\Psi}_{N-3} + (2-\beta_{N-2})\overset{n}{\Psi}_{N-2} - \gamma \overset{n}{\Psi}_{N-1} \\ -\alpha \overset{n}{\Psi}_{N-2} + (2-\beta_{N-1})\overset{n}{\Psi}_{N-1} - \gamma \overset{n}{\Psi}_N - \gamma \overset{n+1}{\Psi}_N \end{bmatrix} \tag{3.116c}$$

and

$$\mathbf{A} = \begin{bmatrix} \beta_1 & \gamma & 0 & 0 & 0 & \cdots & 0 \\ \alpha & \beta_2 & \gamma & 0 & 0 & \cdots & 0 \\ 0 & \alpha & \beta_3 & \gamma & 0 & \cdots & 0 \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & \alpha & \beta_{N-3} & \gamma & 0 \\ 0 & \cdots & 0 & 0 & \alpha & \beta_{N-2} & \gamma \\ 0 & \cdots & 0 & 0 & 0 & \alpha & \beta_{N-1} \end{bmatrix} \tag{3.116d}$$

is a tridiagonal matrix.

Let us first solve a simple problem as a test of our implementation: we will evolve an eigenfunction of the quantum harmonic oscillator to demonstrate that it is a stationary solution. The quantum harmonic oscillator has a potential

$$V(x) = \frac{1}{2}kx^2 \tag{3.117}$$

where k is a spring constant. We use separation of variables to find the stationary eigenfunction. Let

$$\Psi(t,x) = \psi(x)\mathrm{e}^{-iEt/\hbar} \tag{3.118}$$

so that $|\Psi(t,x)|$ is constant in time. Then the time-dependent Schrödinger equation reduces to

$$E\psi(x) = \hat{H}\psi(x) \tag{3.119}$$

which can be rewritten as

$$\frac{2E}{\hbar\omega_0}\,\psi = -\frac{\hbar}{m\omega_0}\frac{d^2\psi}{dx^2} + \frac{m\omega_0}{\hbar}\,x^2\,\psi \tag{3.120}$$

with

$$\omega_0^2 = k/m. \tag{3.121}$$

Now we can construct a dimensionless independent variable

$$y = \sqrt{\frac{m\omega_0}{\hbar}}\,x \tag{3.122}$$

and so obtain the differential equation

$$\frac{d^2\psi}{dy^2} = (y^2 - \lambda)\psi \tag{3.123}$$

with

$$\lambda = \frac{2E}{\hbar\omega_0}. \tag{3.124}$$

To make progress, notice that for $y \to \pm\infty$, the right hand side of Eq. (3.123) $\sim y^2\psi$. Since

$$\frac{d^2}{dy^2} e^{\pm y^2/2} \sim y^2 e^{\pm y^2/2} \tag{3.125}$$

we seek a solution $\psi \sim e^{\pm y^2/2}$. Furthermore, we want our solution to be bounded, so that $\psi \to 0$ as $y \to \pm\infty$, so we write

$$\psi(y) = h(y) e^{-y^2/2} \tag{3.126}$$

where the function $h(y)$ must satisfy the Hermite differential equation

$$h'' - 2yh' + (\lambda - 1)h = 0. \tag{3.127}$$

If we express $h(y)$ as a power series

$$h(y) = \sum_{n=0}^{\infty} c_n y^n \tag{3.128}$$

then Eq. (3.127) becomes

$$\sum_{n=2}^{\infty} n(n-1)c_n y^{n-2} - 2\sum_{n=1}^{\infty} n\, c_n y^n + (\lambda - 1)\sum_{n=0}^{\infty} c_n y^n = 0 \tag{3.129}$$

or, by letting $n \to n+2$ in the first summation,

$$\sum_{n=0}^{\infty} \{[(n+2)(n+1)c_{n+2} - 2n\, c_n + (\lambda - 1)c_n]\, y^n\} = 0. \tag{3.130}$$

Each power of y in this equation must individually vanish which results in the recurrence relationship

$$c_{n+2} = \frac{2n + 1 - \lambda}{(n+1)(n+2)}\, c_n \tag{3.131}$$

where c_0 and c_1 are arbitrary. The solutions then are of the form

$$h(y) = c_0 p(y) + c_1 q(y) \tag{3.132}$$

with

$$\begin{aligned} p(y) &= 1 + \frac{1-\lambda}{2!}\, y^2 + \frac{(1-\lambda)(5-\lambda)}{4!}\, y^4 + \cdots \\ q(y) &= y + \frac{3-\lambda}{3!}\, y^3 + \frac{(3-\lambda)(7-\lambda)}{5!}\, y^5 + \cdots . \end{aligned} \tag{3.133}$$

As $n \to \infty$, we have

$$\frac{c_{n+2}}{c_n} \sim \frac{2}{n} \tag{3.134}$$

and therefore

$$p(y) \sim e^{y^2} \quad \text{and} \quad q(y) \sim e^{y^2} \tag{3.135}$$

so the resulting solution is once again the diverging solution $\psi(y) \sim e^{y^2/2}$. The only way to avoid this is to have one of the c_n coefficients vanish. Then if $c_n = 0$ for some value of n, we see from the recurrence relationship that $c_{n+2} = 0$, $c_{n+4} = 0$, etc., so one functions $p(y)$ or $q(y)$ will be a polynomial of finite order. That is, if $c_n = 0$ for some even value of n then $p(y)$ is a polynomial of finite order and we can choose $c_1 = 0$ to obtain the solution

$$\psi(y) = c_0 p(y) e^{-y^2/2} \tag{3.136}$$

while if $c_n = 0$ for some odd value of n then $q(y)$ is a polynomial of finite order and we can choose $c_0 = 0$ to obtain the solution

$$\psi(y) = c_1 q(y) e^{-y^2/2}. \tag{3.137}$$

The requirement for c_{n+2} to vanish is simply

$$\lambda = \lambda_n = 2n + 1 \tag{3.138}$$

so we can only have bounded solutions when λ is a positive odd integer. The values of λ in which bounded solutions are allowed are known as *eigenvalues*, and their corresponding solutions are *eigenfunctions*. From the relationship between λ and E we obtain the *eigenenergies*

$$E_n = \left(n + \tfrac{1}{2}\right)\hbar\omega_0 \tag{3.139}$$

and the *eigenstates* are then

$$\psi_n(x) \propto \mathscr{H}_n\left(\sqrt{\frac{m\omega_0}{\hbar}}\, x\right) \exp\left(-\frac{1}{2}\frac{m\omega_0}{\hbar}\, x^2\right) \tag{3.140}$$

where the *Hermite polynomial* $\mathscr{H}_n(y)$ is either the polynomial $p(y)$ above if n is even or the polynomial $q(y)$ above if n is odd (see Table 3.1).

For simplicity, we will consider the ground eigenstate

$$\psi_0 = \left(\frac{1}{\pi}\frac{m\omega_0}{\hbar}\right)^{1/4} \exp\left(-\frac{1}{2}\frac{m\omega_0}{\hbar}\, x^2\right) \tag{3.142}$$

belonging to the eigenenergy $E_0 = \frac{1}{2}\hbar\omega_0$. This eigenfunction should be a stationary solution to our numerical implementation of the time-dependent Schrödinger equation and it is: Fig. 3.9 shows that initial data is unchanged by the evolution. The program `oscillator.py` performs the evolution of the wave function under the time-dependent Schrödinger wave equation using the Crank-Nicolson scheme with the harmonic oscillator potential of Eq. (3.117) and initial data corresponding to the ground state given by Eq. (3.142). For convenience, $\hbar$, m, and k are all set to unity. This has the effect of expressing t in units of $(m/k)^{1/2}$ and x in units of $\hbar^{1/2}/(mk)^{1/4}$.

$$\mathscr{H}_0(x) = 1 \tag{3.141a}$$
$$\mathscr{H}_1(x) = 2x \tag{3.141b}$$
$$\mathscr{H}_2(x) = 4x^2 - 2 \tag{3.141c}$$
$$\mathscr{H}_3(x) = 8x^3 - 12x \tag{3.141d}$$
$$\mathscr{H}_4(x) = 16x^4 - 48x^2 + 12 \tag{3.141e}$$
$$\mathscr{H}_5(x) = 32x^5 - 160x^3 + 120x \tag{3.141f}$$
$$\mathscr{H}_{n+1}(x) = 2x\mathscr{H}_n(x) - 2n\mathscr{H}_{n-1}(x) \tag{3.141g}$$
$$\exp(2xt - t^2) = \sum_{n=0}^{\infty} \frac{\mathscr{H}_n(x)}{n!}\, t^n \tag{3.141h}$$

Table 3.1: Hermite polynomials.

Listing 3.6: Program `oscillator.py`

```
import math, cmath, pylab

hbar = 1.0   # reduced Planck's constant
m = 1.0  # mass
k = 1.0  # spring constant

# grid and time intervals
dx = 0.01
dt = 0.05
tmax = 10.0
xmin = -5.0
xmax = 5.0
N = int((xmax-xmin)/dx)

# initial data
x = [0.0]*(N+1)
u = [0.0]*(N+1)
v = [0.0]*(N+1)
p = [0.0]*(N+1)
omega0 = (k/m)**0.5
sigma = (hbar/(2.0*m*omega0))**0.5
for j in range(N+1):
    x[j] = xmin+j*dx
    u[j] = math.exp(-x[j]**2/(4.0*sigma**2))
    u[j] = u[j]/(2.0*math.pi*sigma**2)**0.25
    p[j] = u[j].real**2+u[j].imag**2

# potential
E0 = 0.5*hbar*omega0
V = [0.0]*(N+1)
```

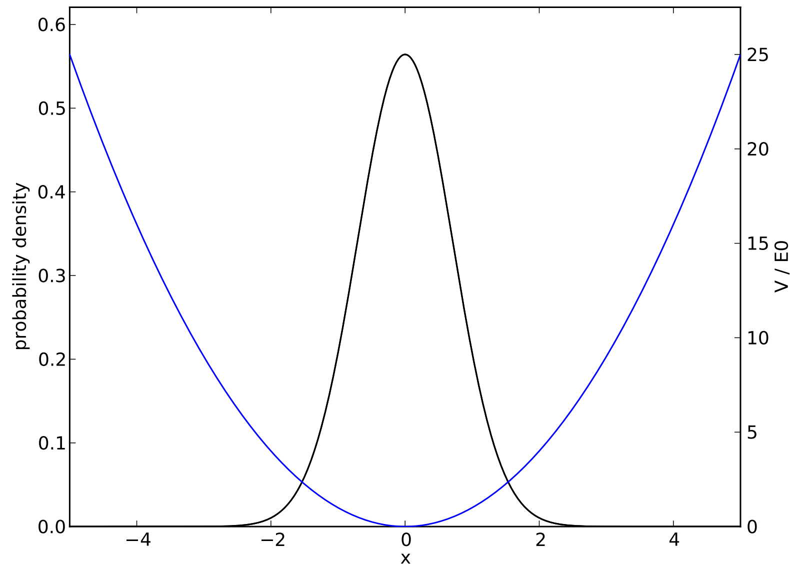

Figure 3.9: Results from running the program `oscillator.py` (Listing 3.6) to evolve the quantum oscillator with initial data corresponding to the ground state. There are two black curves superimposed: the probability density $|\Psi(t,x)|^2$ for the initial data, $t = 0$, and the final value at the end of the run, $t = 10$. The initial and final data are indistinguishable. Also shown in blue is the potential function in units of the ground state energy $V(x)/E_0$. Units of t are in $(m/k)^{1/2}$ while units of x are in $\hbar^{1/2}/(mk)^{1/4}$.

```
31  for j in range(N+1):
32      V[j] = 0.5*k*x[j]**2
33
34
35  # setup coefficients of the tridiagonal matrix
36  alpha = gamma = -1j*hbar*dt/(4.0*m*dx**2)
37  beta = [0.0]*(N+1)
38  for j in range(N):
39      beta[j] = 1.0-2.0*alpha+1j*(V[j]/(2.0*hbar))*dt
40
41  # prepare animated plot
42  pylab.ion()
43  fig = pylab.figure()
44  ax1 = fig.add_subplot(111)
45  ax1.set_xlim(xmin, xmax)
46  ax1.set_ylim(0.0, 1.1*max(p))
47  ax1.set_xlabel('x')
```

```
48 ax1.set_ylabel('probability density')
49 ax2 = ax1.twinx()
50 ax2.set_xlim(xmin, xmax)
51 ax2.set_ylim(0.0, 1.1*max(V)/E0)
52 ax2.set_ylabel('V / E0')
53
54 # plot potential function and wave function
55 ax2.plot(x, [Vj/E0 for Vj in V], 'b')
56 (line, ) = ax1.plot(x, p, 'k-')
57
58 # preform the evolution
59 t = 0.0
60 while t-tmax < 0.5*dt:
61     # update plot
62     for j in range(N+1):
63         p[j] = u[j].real**2+u[j].imag**2
64     line.set_ydata(p)
65     pylab.title('t = %5f'%t)
66     pylab.draw()
67
68     # set the values of the rhs
69     for j in range(1, N):
70         v[j] = -alpha*u[j-1]+(2.0-beta[j])*u[j]-gamma*u[j+1]
71     v[1] -= alpha*u[0]
72     v[N-1] -= gamma*u[N]
73
74     # forward sweep
75     u[1] = v[1]/beta[1]
76     v[1] = gamma/beta[1]
77     for j in range(2, N):
78         den = beta[j]-alpha*v[j-1]
79         u[j] = (v[j]-alpha*u[j-1])/den
80         v[j] = gamma/den
81     # backward sweep
82     for j in reversed(range(1, N)):
83         u[j] -= u[j+1]*v[j]
84     t += dt
85
86 # freeze final plot
87 pylab.ioff()
88 pylab.draw()
89 pylab.show()
```

Now we consider a more interesting problem: the quantum tunneling through a finite potential barrier. Now our potential will be

$$V(x) = \begin{cases} V_0 & \text{for } -a < x < a \\ 0 & \text{otherwise.} \end{cases} \tag{3.143}$$

where a is some constant that describes the half-width of the barrier. Our initial

data will be a Gaussian wave packet moving with group velocity $v_0 = \hbar k_0/m$ where k_0 is the wave vector,

$$\Psi(0,x) = \frac{1}{\sqrt[4]{2\pi\sigma^2}} \mathrm{e}^{-(x-x_0)^2/(2\sigma)^2} \mathrm{e}^{ik_0 x}. \tag{3.144}$$

For this initial data,

$$\langle x \rangle = \int_{-\infty}^{\infty} \Psi^* \, \hat{x} \, \Psi \, dx = x_0 \tag{3.145}$$

and

$$\langle p \rangle = \int_{-\infty}^{\infty} \Psi^* \, \hat{p} \, \Psi \, dx = \hbar k_0 \tag{3.146}$$

where $\hat{x} = x$ and $\hat{p} = -i\hbar(\partial/\partial x)$. In order for our wave packet to have a well-defined momentum, we require $\sigma \gg 1/k_0$, and in order to resolve the wave packet on our grid we require $\Delta x \ll \sigma$. Additionally, the wavelength corresponding to wave number k_0 is $\lambda_0 = 2\pi/k_0$ and if we require a sufficiently fine grid to resolve this, so $\Delta x \ll 1/k_0$. We are thus left with the requirement $k_0 \Delta x \ll 1 \ll k_0 \sigma$. To *accurately* monitor the evolution, we also want $v_0 \Delta t \le \Delta x$ or $\Delta t \le m \Delta x / \hbar k_0$. The initial energy of the packet is approximately $E_0 = \frac{1}{2} m v_0^2 = \hbar^2 k_0^2 / 2m$, and to allow for some partial tunneling we will choose $V_0 \sim E_0$ and $a \sim 1/k_0$.

The code `tunnel.py` evolves this system with the parameters $V_0 = \hbar^2 k_0^2/2m$, $k_0 a = 5$, $k_0 \sigma = 10$, $k_0 x_0 = -100$, and $k_0 \Delta x = \hbar k_0^2 \Delta t / m = \frac{1}{4}$. For convenience, units are chosen so that $\hbar = 1$, $m = 1$, and $k_0 = 1$. The code is identical to `oscillator.py` except in the beginning where the grid, the initial data, and the potential are created. The initial part of `tunnel.py` that differs from `oscillator.py` is given in Listing 3.7 and the results produced by the program are shown in Fig. 3.10. We see that most of the wave packet is reflected off of the potential barrier but that some portion of it tunnels through the barrier.

Listing 3.7: Modified lines in program `tunnel.py`

```
3  hbar = 1.0  # reduced Planck's constant
4  m = 1.0  # mass
5  k0 = 1.0  # initial wavenumber
6
7  # grid and time intervals
8  dx = 0.25/k0
9  dt = 0.25*m/(hbar*k0**2)
10 tmax = 200.0*m/(hbar*k0**2)
11 xmin = -200.0/k0
12 xmax = 200.0/k0
13 N = int((xmax-xmin)/dx)
14
15 # initial data
16 x = [0.0]*(N+1)
17 u = [0.0]*(N+1)
18 v = [0.0]*(N+1)
```

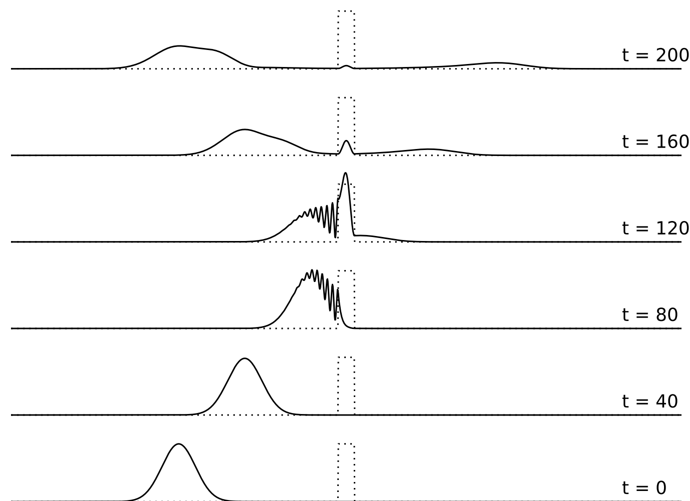

Figure 3.10: Results from running the program `tunnel.py` (Listing 3.7). The dotted lines show the potential barrier which has width $2a = 10/k_0$ and height $V_0 = \hbar^2 k_0^2/2m$ where $k_0 = \langle p \rangle/\hbar$ is the average wave number of the initial wave packet, which had width $\sigma = 10/k_0$.

```
19 p = [0.0]*(N+1)
20 x0 = -100.0/k0
21 sigma = 10.0/k0
22 for j in range(N+1):
23     x[j] = xmin+j*dx
24     u[j] = cmath.exp(-(x[j]-x0)**2/(4.0*sigma**2)+1j*k0*x[j])
25     u[j] = u[j]/(2.0*math.pi*sigma**2)**0.25
26     p[j] = u[j].real**2+u[j].imag**2
27
28 # potential
29 E0 = (hbar*k0)**2/(2.0*m)
30 a = 5.0/k0
31 V = [0.0]*(N+1)
32 for j in range(N+1):
33     if abs(x[j]) < a: V[j] = E0
```

Exercise 3.3 Investigate the quantum scattering off of a potential shelf

$$V(x) = \begin{cases} 0 & \text{for } x \le 0 \\ -\alpha p_0^2/2m & \text{for } x > 0 \end{cases} \tag{3.147}$$

for a wave packet initially centered around x_0 with $x_0 < 0$ and moving in the positive-x direction with momentum $p_0 = \hbar k_0$. Here α is a positive constant. Find the probability of the particle reflecting off of the shelf for various values of α.

Now consider the Schrödinger equation in more than one spatial dimensions,

$$i\hbar \frac{\partial \Psi}{\partial t} = -\frac{\hbar^2}{2m}\nabla^2\Psi + V(\boldsymbol{x})\Psi, \tag{3.148}$$

where $V(\boldsymbol{x})$ is the potential and $\Psi(t, \boldsymbol{x})$ is the wave function. The solution is

$$\Psi(t, \boldsymbol{x}) = \mathrm{e}^{-i\hat{H}t/\hbar}\Psi(0, \boldsymbol{x}) \tag{3.149}$$

with

$$\hat{H} = -\frac{\hbar^2}{2m}\nabla^2 + V(\boldsymbol{x}). \tag{3.150}$$

Employing Cayley's form of the unitary operator and restricting ourselves to two spatial dimensions, we find

$$(1 + \tfrac{1}{2}i\hat{H}\Delta t/\hbar)\overset{n+1}{\Psi}_{j,k} = (1 - \tfrac{1}{2}i\hat{H}\Delta t/\hbar)\overset{n}{\Psi}_{j,k} \tag{3.151}$$

where

$$\hat{H}\overset{n}{\Psi}_{j,k} = -\frac{\hbar^2}{2m}\frac{\overset{n}{\Psi}_{j+1,k} - 2\overset{n}{\Psi}_{j,k} + \overset{n}{\Psi}_{j-1,k}}{(\Delta x^2)} - \frac{\hbar^2}{2m}\frac{\overset{n}{\Psi}_{j,k+1} - 2\overset{n}{\Psi}_{j,k} + \overset{n}{\Psi}_{j,k-1}}{(\Delta y^2)} + V_{j,k}\overset{n}{\Psi}_{j,k} \tag{3.152}$$

and

$$\overset{n}{\Psi}_{j,k} = \Psi(n\,\Delta t, j\,\Delta x, k\,\Delta y). \tag{3.153}$$

The computation of $\overset{n+1}{\Psi}_{j,k}$ given $\overset{n}{\Psi}_{j,k}$ can again be expressed in the form of the linear problem $\mathbf{A} \cdot \boldsymbol{x} = \boldsymbol{b}$, but now the matrix $\mathbf{A}$ is not tri-diagonal (though it is sparse).

As an alternative, let us consider a method known as *operator splitting*. Notice that the Hamiltonian operator can be split into the sum of two Hamiltonian operators,

$$\hat{H} = \hat{H}_x + \hat{H}_y \tag{3.154}$$

(we continue to restrict ourselves to two spatial dimensions) where

$$\hat{H}_x = -\frac{\hbar^2}{2m}\frac{\partial^2}{\partial x^2} + \frac{1}{2}V(x, y) \tag{3.155}$$

and

$$\hat{H}_y = -\frac{\hbar^2}{2m}\frac{\partial^2}{\partial y^2} + \frac{1}{2}V(x,y). \tag{3.156}$$

The finite difference equations are now

$$(1 + \tfrac{1}{2}i\hat{H}_y\Delta t/\hbar)(1 + \tfrac{1}{2}i\hat{H}_x\Delta t/\hbar)\overset{n+1}{\Psi}_{j,k} = (1 - \tfrac{1}{2}i\hat{H}_x\Delta t/\hbar)(1 - \tfrac{1}{2}i\hat{H}_y\Delta t/\hbar)\overset{n}{\Psi}_{j,k}\,. \tag{3.157}$$

This is equivalent to the system of equations

$$(1 + \tfrac{1}{2}i\hat{H}_x\Delta t/\hbar)\Xi_{j,k} = (1 - \tfrac{1}{2}i\hat{H}_y\Delta t/\hbar)\overset{n}{\Psi}_{j,k} \tag{3.158}$$

$$(1 + \tfrac{1}{2}i\hat{H}_y\Delta t/\hbar)\overset{n+1}{\Psi}_{j,k} = (1 - \tfrac{1}{2}i\hat{H}_x\Delta t/\hbar)\Xi_{j,k}. \tag{3.159}$$

Here the auxiliary field Ξ is the "mid-step" value of Ψ. Both of these equations can be written as a linear system with tri-diagonal matrices.

We illustrate this method with the problem of hard-"sphere" scattering in two dimensions: an initial wave packet travels toward a potential

$$V(x,y) = \begin{cases} V_0 & \text{for } x^2 + y^2 < a^2 \\ 0 & \text{otherwise} \end{cases} \tag{3.160}$$

where a is the radius of the circular potential and V_0 is the height of the potential. We choose $k_0 a > 1$ so that we are in the short-wavelength regime, and we choose V_0 to be comparable to the energy of the particle. The program `scatter.py` is listed below, and the results can be seen in Fig. 3.11.

Listing 3.8: Program `scatter.py`

```
import math, cmath, pylab

# solves the tridiagonal system of equations A.x = b
def tridiag(alp, bet, gam, b):
    n = len(bet)
    x = pylab.zeros(b.shape, dtype=complex)
    y = pylab.zeros(b.shape, dtype=complex)
    y[0] = gam[0]/bet[0]
    x[0] = b[0]/bet[0]
    for i in range(1, n):
        den = bet[i]-alp[i]*y[i-1]
        y[i] = gam[i]/den
        x[i] = (b[i]-alp[i]*x[i-1])/den
    for i in reversed(range(n-1)):
        x[i] -= x[i+1]*y[i]
    return x


hbar = 1.0  # reduced Planck's constant
m = 1.0  # mass
k0 = 1.0  # initial wavenumber
```

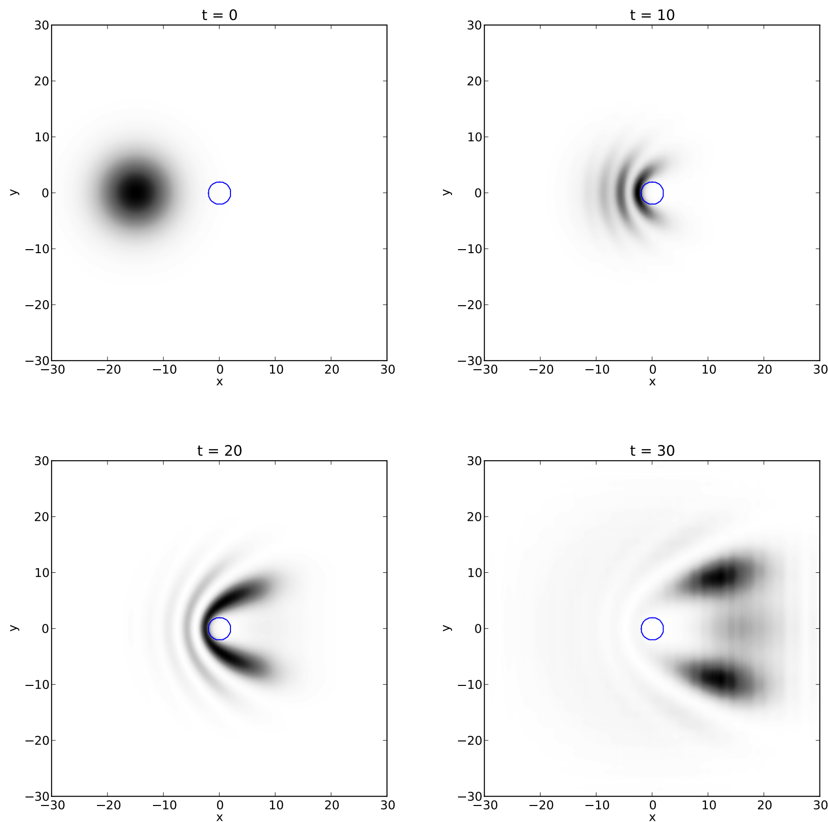

Figure 3.11: Results from running the program `scatter.py` (Listing 3.8) but at a finer resolution, $\Delta x = \Delta y = 0.1/k_0$ and $\Delta t = 0.1m/(\hbar k_0^2)$, than in the listing. The blue circle contour indicates the region where the potential is non-zero.

```

# grid and time intervals
dy = dx = 0.5/k0
dt = 0.5*m/(hbar*k0**2)
tmax = 30.0*m/(hbar*k0**2)
ymin = xmin = -30.0/k0
ymax = xmax = 30.0/k0

# initial data
x = pylab.arange(xmin, xmax, dx)
y = pylab.arange(ymin, ymax, dy)
N = len(x)
u = pylab.zeros((N, N), dtype=complex)
```

```
35 v = pylab.zeros((N, N), dtype=complex)
36 p = pylab.zeros((N, N))
37 x0 = -15.0/k0
38 y0 = 0.0/k0
39 sigma = 5.0/k0
40 for j in range(N):
41     for k in range(N):
42         rr = (x[j]-x0)**2+(y[k]-y0)**2
43         u[j,k] = cmath.exp(-rr/(4.0*sigma**2)+1j*k0*x[j])
44         p[j,k] = u[j,k].real**2+u[j,k].imag**2
45
46 # potential
47 a = 2.0/k0
48 E0 = (hbar*k0)**2/(2.0*m)
49 V = pylab.zeros((N, N))
50 for j in range(N):
51     for k in range(N):
52         rr = x[j]**2+y[k]**2
53         if rr < a**2:
54             V[j,k] = E0
55
56 # prepare animated plot
57 pylab.ion()
58 pylab.xlabel('x')
59 pylab.ylabel('y')
60 image = pylab.imshow(p.T, origin='upper', extent=(xmin, xmax, ymin, ymax
       ),
61                      cmap=pylab.cm.hot)
62 pylab.contour(V.T, origin='upper', extent=(xmin, xmax, ymin, ymax),
       colors='c')
63
64 # setup coefficients of the tridiagonal matrix
65 alpha = gamma = -1j*hbar*dt/(4.0*m*dx**2)
66 alp = alpha*pylab.ones(N, dtype=complex)
67 gam = gamma*pylab.ones(N, dtype=complex)
68 bet = pylab.zeros((N, N), dtype=complex)
69 for j in range(N):
70     for k in range(N):
71         bet[j,k] = 1.0-2.0*alpha+1j*(V[j,k]/(2.0*hbar))*dt
72
73 # preform the evolution; blithely ignore boundary conditions
74 t = 0.0
75 while t-tmax < 0.5*dt:
76
77     # update plot
78     for j in range(N):
79         for k in range(N):
80             p[j,k] = u[j,k].real**2+u[j,k].imag**2
81     image.set_data(p.T)
82     pylab.title('t = %5f'%t)
```

```
83        pylab.draw()
84
85        # first step of operator splitting
86        for j in range(1, N-1):
87            for k in range(1, N-1):
88                v[j,k] = -alpha*u[j,k-1]+(2.0-bet[j,k])*u[j,k]-gamma*u[j,k
          +1]
89        for k in range(1, N-1):
90            u[:,k] = tridiag(alp, bet[:,k], gam, v[:,k])
91
92        # second step of operator splitting
93        for k in range(1, N-1):
94            for j in range(1, N-1):
95                v[j,k] = -alp[j]*u[j-1,k]+(2.0-bet[j,k])*u[j,k]-gam[j]*u[j
          +1,k]
96        for j in range(1, N-1):
97            u[j,:] = tridiag(alp, bet[j,:], gam, v[j,:])
98        t += dt
99
100   # freeze final plot
101   pylab.ioff()
102   pylab.draw()
103   pylab.show()
```

3.4 Electric potentials

We now consider elliptic equations. The Poisson equation in three dimensions is

$$\frac{\partial^2 u(x,y,z)}{\partial x^2} + \frac{\partial^2 u(x,y,z)}{\partial y^2} + \frac{\partial^2 u(x,y,z)}{\partial z^2} = -\frac{\rho(x,y,z)}{\epsilon_0} \tag{3.161}$$

where $u(x,y,z)$ is the electric potential field and $\rho(x,y,z)$ is the charge density. For simplicity, though, we will investigate the Laplace equation in two dimensions

$$\frac{\partial^2 u(x,y)}{\partial x^2} + \frac{\partial^2 u(x,y)}{\partial y^2} = 0 \tag{3.162}$$

on the square $0 \le x \le L$ and $0 \le y \le L$ with one wall of the square kept (the wall at $y = L$) at a potential of $V_0 = 1\,\mathrm{V}$ and the other walls grounded at $0\,\mathrm{V}$.

We use separation of variables

$$u(x,y) = X(x)Y(y) \tag{3.163}$$

to express Laplace's equation as the ordinary differential equations

$$-\frac{X''}{X} = \frac{Y''}{Y} = k^2 \tag{3.164}$$

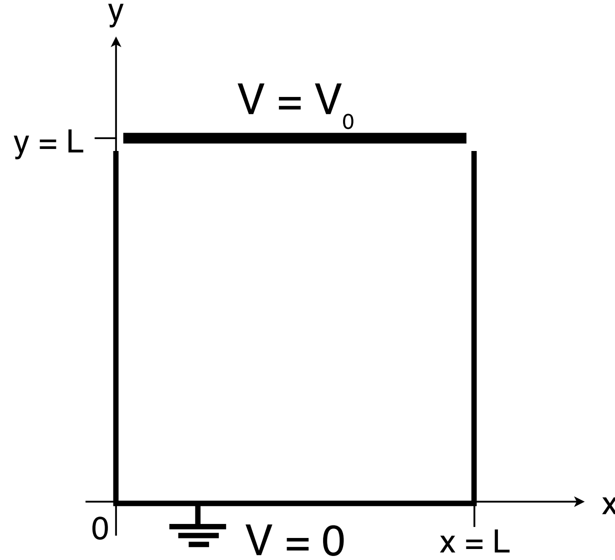

Figure 3.12: Example boundary value problem for Laplace's equation in two dimensions.

where k is a separation constant. The boundary conditions are $X(0) = 0$, $X(L) = 0$, $Y(0) = 0$, and $Y(L) = V_0$. The solutions for X with these boundary conditions are

$$X(x) \propto \sin \frac{n\pi x}{L} \qquad \text{for } n = 1, 2, 3, \ldots \tag{3.165}$$

and the allowed separation constants are $k_n = n\pi/L$. The solutions for Y will be a linear combination of hyperbolic sine and hyperbolic cosine functions, but since only the hyperbolic sine function vanishes at $x = 0$ our solution is of the form

$$u(x, y) = \sum_{n=1}^{\infty} c_n \sin \frac{n\pi x}{L} \sinh \frac{n\pi y}{L}. \tag{3.166}$$

The final boundary condition, $u(x, L) = V_0$, then determines the coefficients c_n: we have

$$\sum_{n=1}^{\infty} c_n \sinh n\pi \sin \frac{n\pi x}{L} = V_0. \tag{3.167}$$

We multiply both sides by $\sin(m\pi x/L)$ and integrate from $x = 0$ to $x = L$ to obtain

$$\frac{L}{2} c_m \sinh m\pi = \begin{cases} \dfrac{2LV_0}{m\pi} & \text{for } m \text{ odd} \\ 0 & \text{otherwise.} \end{cases} \tag{3.168}$$

We thus have

$$c_m = \begin{cases} \dfrac{4V_0}{m\pi \sinh m\pi} & \text{for } m \text{ odd} \\ 0 & \text{otherwise} \end{cases} \tag{3.169}$$

and the solution to Laplace's equation is

$$u(x,y) = 4V_0 \sum_{\substack{n=1 \\ n \text{ odd}}}^{\infty} \frac{\sin(n\pi x/L)}{n\pi} \frac{\sinh(n\pi y/L)}{\sinh n\pi}. \tag{3.170}$$

A large number of terms in this series are needed in order to accurately compute the field near the wall near $y = L$ (and especially at the corners).

The *relaxation method* can be used to elliptic equations of the form

$$\hat{L}u = \rho \tag{3.171}$$

where $\hat{L}$ is an elliptic operator and ρ is a source term. The approach is to take an initial distribution u that does not necessarily solve the elliptic equation and allow it to relax to the solution of the equation by evolving the diffusion equation

$$\frac{\partial u}{\partial t} = \hat{L}u - \rho. \tag{3.172}$$

At late times, $t \to 0$, the solution will asymptotically approach the stationary solution to the elliptic equation.

For the problem at hand ($\rho = 0$ and $\hat{L}$ is the two-dimensional Laplacian operator) the FTCS method applied to the diffusion equation results in

$$\overset{n+1}{u}_{j,k} = \overset{n}{u}_{j,k} + \left[\frac{\overset{n}{u}_{j+1,k} - 2\,\overset{n}{u}_{j,k} + \overset{n}{u}_{j-1,k}}{(\Delta x)^2} + \frac{\overset{n}{u}_{j,k+1} - 2\,\overset{n}{u}_{j,k} + \overset{n}{u}_{j,k-1}}{(\Delta y)^2} \right] \Delta t \tag{3.173}$$

where $\overset{n}{u}_{j,k} = \overset{n}{u}\,(j\,\Delta x, k\,\Delta y)$ and n indicates the iteration. For simplicity we take $\Delta x = \Delta y = \Delta$ so we have

$$\overset{n+1}{u}_{j,k} = (1-\omega)\,\overset{n}{u}_{j,k} + \frac{\omega}{4}\left(\overset{n}{u}_{j+1,k} + \overset{n}{u}_{j-1,k} + \overset{n}{u}_{j,k+1} + \overset{n}{u}_{j,k-1}\right) \tag{3.174}$$

where $\omega = 4\Delta t/\Delta^2$.

Stability of the diffusion equation limits the magnitude of ω. We can determine the maximum value of ω using a von Neumann stability analysis, but now we will limit ourselves to the spatial eigenmodes that satisfy the Dirichlet boundary conditions for the homogeneous part of the solution. Our ansatz is therefore

$$\overset{n}{u}_{j,k} = \overset{0}{u}\,\xi^n(m_x, m_y) \sin\frac{m_x \pi j \Delta}{L} \sin\frac{m_y \pi k \Delta}{L} \tag{3.175}$$

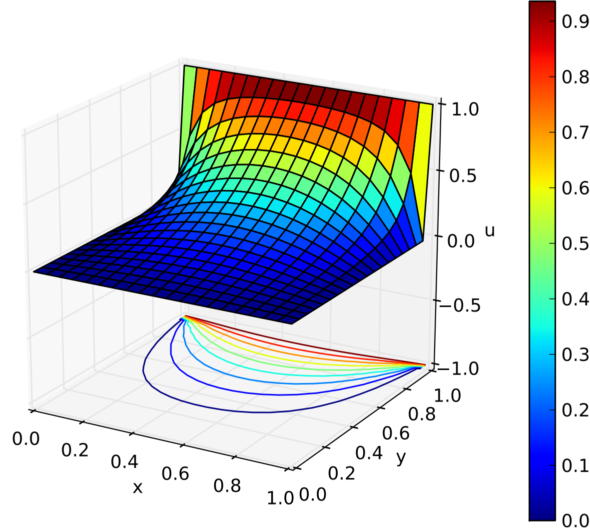

Figure 3.13: Results from running the program `relax.py` (Listing 3.9) with parameters $N = 20$ grid points along each side. The solution is obtained after 386 iterations.

where m_x and m_y are the mode numbers in the x and y direction respectively. By substituting this ansatz into Eq. (3.174) we find

$$\xi(m_x, m_y) = 1 - \omega + \frac{\omega}{2}\left(\cos\frac{m_x \pi \Delta}{L} + \cos\frac{m_y \pi \Delta}{L}\right) \tag{3.176}$$

and we see that stability is achieved, i.e., that $|\xi(m_x, m_y)| \leq 1$ for any mode given by (m_x, m_y), if $\omega \leq 1$. If we now take the largest value of $\Delta t = \Delta^2/4$ allowed for stable iteration corresponding to $\omega = 1$, we obtain the following iteration scheme:

$$\overset{n+1}{u}_{j,k} = \frac{1}{4}\left(\overset{n}{u}_{j+1,k} + \overset{n}{u}_{j-1,k} + \overset{n}{u}_{j,k+1} + \overset{n}{u}_{j,k-1}\right). \tag{3.177}$$

Here we see that the value of the field at a given lattice point (j, k) at step $n + 1$ is equal to the average of the values of the field at the neighboring points at step n. This is known as *Jacobi's method*.

The program `relax.py` implements Jacobi's method for our model problem. The number of grid points along each side, N, is input. Results obtained for $N = 20$ points are shown in Fig. 3.13.

Listing 3.9: Program `relax.py`

```
import math, pylab, mpl_toolkits.mplot3d

eps = 1e-5  # fractional error allowed
L = 1.0  # length of each side
N = input('number of grid points on a side -> ')
dy = dx = L/(N-1.0)
x = pylab.array(range(N))*dx
y = pylab.array(range(N))*dy
(x, y) = pylab.meshgrid(x, y)
u0 = pylab.zeros((N, N))
u1 = pylab.zeros((N, N))

# boundary conditions
for j in range(N):
    u1[j,N-1] = u0[j,N-1] = 1.0

# prepare animated plot
pylab.ion()
image = pylab.imshow(u0.T, origin='lower', extent=(0.0, L, 0.0, L))

n = 0  # number of iterations
err = 1.0  # average error per site
while err > eps:
    # update animated plot
    image.set_data(u0.T)
    pylab.title('iteration %d'%n)
    pylab.draw()

    # next iteration in refinement
    n = n+1
    err = 0.0
    for j in range(1, N-1):
        for k in range(1, N-1):
            u1[j,k] = (u0[j-1,k]+u0[j+1,k]+u0[j,k-1]+u0[j,k+1])/4.0
            err += abs(u1[j,k]-u0[j,k])
    err /= N**2
    (u0, u1) = (u1, u0)  # swap old and new arrays for next iteration

# surface plot of final solution
pylab.ioff()
fig = pylab.figure()
axis = fig.gca(projection='3d', azim=-60, elev=20)
surf = axis.plot_surface(x, y, u0.T, rstride=1, cstride=1, cmap=pylab.cm
    .jet)
axis.contour(x, y, u0.T, 10, zdir='z', offset=-1.0)
axis.set_xlabel('x')
axis.set_ylabel('y')
axis.set_zlabel('u')
axis.set_zlim(-1.0, 1.0)
fig.colorbar(surf)
```

```
50 pylab.show()
```

In the program `relax.py`, the iteration was continued until the average error per mesh point was less than $\epsilon = 10^{-5}$ where the error was estimated by taking the difference between the old value at step n and the new value at step $n+1$. The number of iterations required for convergence depends on the slowest decaying eigenmode of the iteration. The modulus of the slowest-decaying mode is known as the *spectral radius*,

$$\rho = \max_{m_x, m_y} |\xi(m_x, m_y)|. \tag{3.178}$$

From Eq. (3.176) we see that for the Jacobi method with $\omega = 1$ we have

$$\rho = \rho_{\mathrm{J}} = \cos\frac{\pi\Delta}{L} \tag{3.179}$$

which corresponds to the $m_x = m_y = 1$ mode. Each iteration multiplies the residual error in this least-damped mode by a factor with modulus ρ and so the number of iterations required to achieve a desired error tolerance ϵ will be

$$n \approx \frac{\ln\epsilon}{\ln\rho}. \tag{3.180}$$

For the Jacobi method, $\rho_{\mathrm{J}} \approx 1 - \frac{1}{2}(\pi/N)^2$ and so

$$n \approx \frac{2|\ln\epsilon|}{\pi^2} N^2. \tag{3.181}$$

Note that if we double the number of grid points N then we require four times as many iterations to converge. For practical problems, Jacobi's method converges too slowly to be useful.

To make progress, consider again Eq. (3.174) and notice that if we imagine our computational grid to be divided into staggered light and dark points, as depicted in Fig. 3.14 then to update the value of a white point we need only the current value at that white point and the values of the neighboring dark points, and vice-versa to update a value of a dark point. Thus we can take a staggered approach where we update all the white points and then we update all the black points and both steps can be done *in place*. For steps where n is an integer, we compute the values of the white points using the formula

$$\overset{n+1}{u}_{j,k} = (1-\omega)\,\overset{n}{u}_{j,k} + \frac{\omega}{4}\left(\overset{n+1/2}{u}_{j+1,k} + \overset{n+1/2}{u}_{j-1,k} + \overset{n+1/2}{u}_{j,k+1} + \overset{n+1/2}{u}_{j,k-1}\right) \tag{3.182}$$

and then for steps where n is a half-integer we use the same formulate to compute the values of the black points. We can repeat the stability analysis using the ansatz of Eq. (3.175) and we find

$$\xi^{1/2}(m_x, m_y) = \frac{\omega c \pm \sqrt{\omega^2 c^2 - 4(\omega - 1)}}{2} \tag{3.183}$$

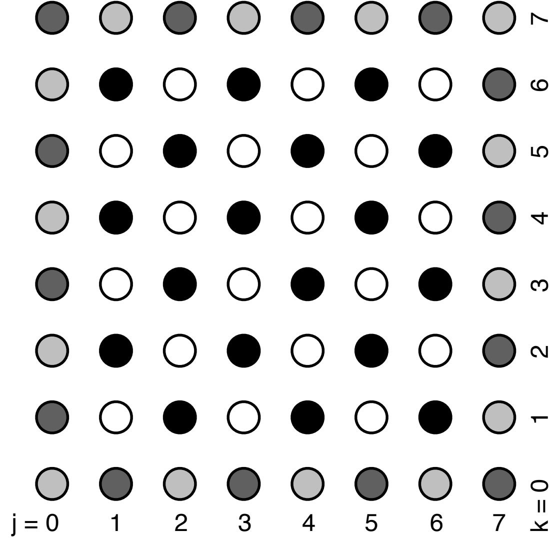

Figure 3.14: A staggered lattice of dark and light points for use in successive over-relaxation. The grey points are part of the boundary and are not evolved. To update a white point requires only the previous value of the white point and the surrounding dark points and similarly to update a black point requires only the previous value of the black points and the surrounding light points.

where

$$c = \frac{1}{2}\left(\cos\frac{m_x \pi \Delta}{L} + \cos\frac{m_x \pi \Delta}{L}\right). \tag{3.184}$$

This reveals that scheme of Eq. 3.189 is stable for $0 < \omega < 2$. When $\omega = 1$ the method is known as the *Gauss-Seidel method*, which converges somewhat faster than the Jacobi method. For $\omega > 1$ we have accelerated convergence (relative to relaxation) which is known as *successive over-relaxation* or SOR. The parameter ω is known as the *over-relaxation parameter*.

There is an optimal value for the over-relaxation parameter for which the spectral radius is minimized. If we focus on the least-damped mode for which $m_x = m_y = 1$, we have $c = \rho_{\mathrm{J}}$ and so the spectral radius as a function of ω can be written as

$$\rho(\omega) = \begin{cases} \left[\frac{1}{2}\omega\rho_{\mathrm{J}} + \frac{1}{2}\sqrt{\omega^2\rho_{\mathrm{J}}^2 - 4(\omega-1)}\right]^2 & \text{for } 0 < \omega \le \omega_{\mathrm{opt}} \\ \omega - 1 & \text{for } \omega_{\mathrm{opt}} \le \omega < 2 \end{cases} \tag{3.185}$$

where ω_{opt} is the optimal choice that minimizes ρ,

$$\begin{aligned}\omega_{\mathrm{opt}} &= \frac{2}{1+\sqrt{1-\rho_{\mathrm{J}}^2}} \\ &= \frac{2}{1+\sin(\pi\Delta/L)} \quad \text{for } \rho_{\mathrm{J}} = \cos(\pi\Delta/L).\end{aligned} \tag{3.186}$$

Thus, for the optimal choice of the over-relaxation parameter, $\omega = \omega_{\mathrm{opt}} \approx \pi/N$, the spectral radius is

$$\rho(\omega_{\mathrm{opt}}) = \rho_{\mathrm{opt}} = \frac{1-\sin(\pi\Delta/L)}{1+\sin(\pi\Delta/L)} \approx 1 - \frac{2\pi}{N} \tag{3.187}$$

and the number of iterations required to reduce the error to some tolerance ϵ is

$$n \approx \frac{\ln\epsilon}{\ln\rho_{\mathrm{opt}}} \approx \frac{|\ln\epsilon|}{2\pi} N. \tag{3.188}$$

Now the number of iterations is proportional to N rather than N^2 so convergence is achieved much more rapidly for large values of N.

The program `overrelax.py` is a modification to `relax.py` that implements successive over-relaxation. The program differs in many places so it is listed in its entirety. The results for $N = 100$ points along a side are displayed in Fig. 3.15. Convergence occurs in 137 iterations.

Listing 3.10: Program `overrelax.py`

```
import math, pylab, mpl_toolkits.mplot3d

eps = 1e-5  # fractional error allowed
L = 1.0  # length of each side
N = input('number of grid points on a side -> ')
dy = dx = L/(N-1.0)
x = pylab.array(range(N))*dx
y = pylab.array(range(N))*dy
(x, y) = pylab.meshgrid(x, y)
u = pylab.zeros((N, N))

# boundary conditions
for j in range(N):
    u[j,N-1] = 1.0

# compute over-relaxation parameter
omega = 2.0/(1.0+math.sin(math.pi*dx/L))

# white and black pixels: white have j+k even; black have j+k odd
white = [(j, k) for j in range(1, N-1) for k in range(1, N-1) if (j+k)
    %2 == 0]
black = [(j, k) for j in range(1, N-1) for k in range(1, N-1) if (j+k)
    %2 == 1]
```

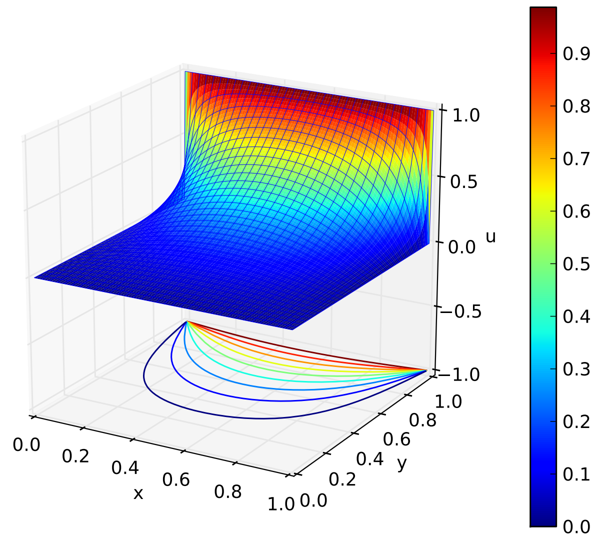

Figure 3.15: Results from running the program `overrelax.py` (Listing 3.10) with parameters $N = 100$ grid points along each side. The solution is obtained after 137 iterations.

```
22
23 # prepare animated plot
24 pylab.ion()
25 image = pylab.imshow(u.T, origin='lower', extent=(0.0, L, 0.0, L))
26
27 n = 0   # number of iterations
28 err = 1.0   # average error per site
29 while err > eps:
30     # update animated plot
31     image.set_data(u.T)
32     pylab.title('iteration %d'%n)
33     pylab.draw()
34
35     # next iteration in refinement
36     n = n+1
37     err = 0.0
38     for (j, k) in white+black:   # loop over white pixels then black pixels
39         du = (u[j-1,k]+u[j+1,k]+u[j,k-1]+u[j,k+1])/4.0-u[j,k]
40         u[j,k] += omega*du
41         err += abs(du)
42     err /= N**2
```

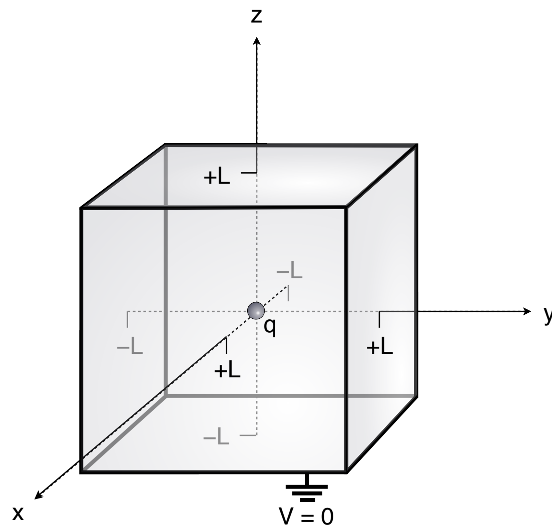

Figure 3.16: Example boundary value problem for Poisson's equation: a point charge q is located in the center of a cube with edge length $2L$ whose faces are grounded.

```
43
44  # surface plot of final solution
45  pylab.ioff()
46  fig = pylab.figure()
47  axis = fig.gca(projection='3d', azim=-60, elev=20)
48  surf = axis.plot_surface(x, y, u.T, rstride=1, cstride=1, linewidth=0,
49                           cmap=pylab.cm.jet)
50  wire = axis.plot_wireframe(x, y, u.T, rstride=1+N//50, cstride=1+N//50,
51                             linewidth=0.25)
52  axis.contour(x, y, u.T, 10, zdir='z', offset=-1.0)
53  axis.set_xlabel('x')
54  axis.set_ylabel('y')
55  axis.set_zlabel('u')
56  axis.set_zlim(-1.0, 1.0)
57  fig.colorbar(surf)
58  pylab.show()
```

As a final example, let us solve for the electric potential produced by a point charge centered in a cube with edges of length $2L$ in which the faces of the cube are grounded as shown in Fig. 3.16. This is now a three-dimensional problem which we can again solve using successive over-relaxation, and our iteration equation now

also includes a source term:

$$\overset{n+1}{u}_{i,j,k} = (1-\omega)\,\overset{n}{u}_{i,j,k} + \frac{\omega}{6}\Big(\overset{n+1/2}{u}_{i+1,j,k} + \overset{n+1/2}{u}_{i-1,j,k} + \overset{n+1/2}{u}_{i,j+1,k} + \overset{n+1/2}{u}_{i,j-1,k} + \overset{n+1/2}{u}_{i,j,k+1} + \overset{n+1/2}{u}_{i,j,k-1}\Big) + \frac{\omega}{6}\frac{\rho_{i,j,k}}{\epsilon_0}\Delta^2. \tag{3.189}$$

Again we divide the lattice of grid points into alternating "white" and "black" pixels and solve for the white pixels on the integer steps and for the black pixels on the half integer steps. The point charge gives us a charge density that we take to be

$$\rho_{i,j,k} = \begin{cases} \dfrac{q}{\Delta^3} & \text{for } i=j=k=(N-1)/2 \\ 0 & \text{otherwise} \end{cases} \tag{3.190}$$

and we must be sure to choose an odd value for N so that there is a grid point at the exact center of the box.

The program `charge.py` listed below computes the electric potential field within the grounded box for a unit $q/\epsilon_0 = 1$ charge. The results are shown in Fig. 3.17.

Listing 3.11: Program `charge.py`

```
1  import math, pylab, mpl_toolkits.mplot3d, matplotlib.colors
2
3  eps = 1e−5   # fractional error allowed
4  L = 1.0   # half−length of each side
5  N = input('number of grid points on a side −> ')
6  dz = dy = dx = 2.0*L/(N−1.0)
7  x = −L+pylab.array(range(N))*dx
8  y = −L+pylab.array(range(N))*dy
9  z = −L+pylab.array(range(N))*dz
10 u = pylab.zeros((N, N, N))
11 rho = pylab.zeros((N, N, N))
12
13 # source
14 q = 1.0
15 rho[(N−1)//2,(N−1)//2,(N−1)//2] = q/(dx*dy*dz)
16
17 # prepare animated plot
18 pylab.ion()
19 s = u[:,:,(N−1)//2]
20 image = pylab.imshow(s.T, origin='lower', extent=(−L, L, −L, L), vmax
       =1.0)
21
22 # compute over−relaxation parameter
23 omega = 2.0/(1.0+math.sin(math.pi*dx/L))
```

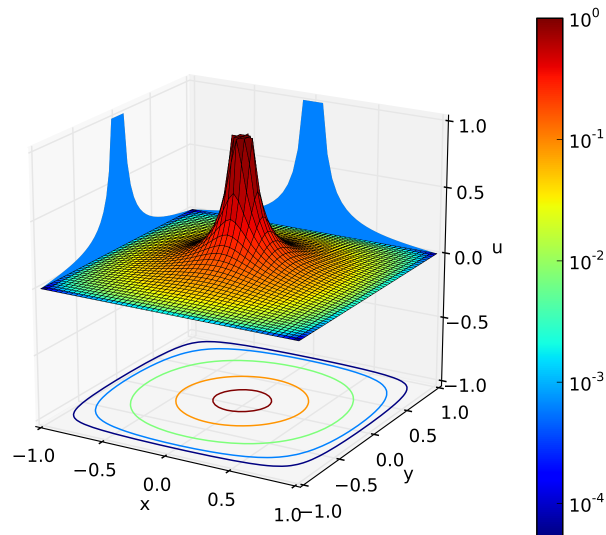

Figure 3.17: Results from running the program `charge.py` (Listing 3.11) with parameters $N = 51$ grid points along each side. The value of the electric potential on the $z = 0$ plane is shown (the colors values are logarithmic in the potential).

```
24
25 # white and black pixels: white have i+j+k even; black have i+j+k odd
26 white = [(i, j, k) for i in range(1, N−1) for j in range(1, N−1) for k
       in
27         range(1, N−1) if (i+j+k)%2 == 0]
28 black = [(i, j, k) for i in range(1, N−1) for j in range(1, N−1) for k
       in
29         range(1, N−1) if (i+j+k)%2 == 1]
30 n = 0   # number of iterations
31 err = 1.0   # average error per site
32 while err > eps:
33     image.set_data(s.T)
34     pylab.title('iteration %d'%n)
35     pylab.draw()
36
37     # next iteration in refinement
38     n = n+1
39     err = 0.0
40     for (i, j, k) in white+black:   # loop over white pixels then black pixels
41         du = (u[i−1,j,k]+u[i+1,j,k]+u[i,j−1,k]+u[i,j+1,k]+u[i,j,k−1]+u[
```

```
    i,j,k+1]
42              +dx**2*rho[i,j,k])/6.0-u[i,j,k]
43        u[i,j,k] += omega*du
44        err += abs(du)
45    err /= N**3
46
47 # surface plot of final solution
48 (x, y) = pylab.meshgrid(x, y)
49 s = s.clip(eps, 1.0)
50 levels = [10**(l/2.0) for l in range(-5, 0)]
51 pylab.ioff()
52 fig = pylab.figure()
53 axis = fig.gca(projection='3d', azim=-60, elev=20)
54 surf = axis.plot_surface(x, y, s.T, rstride=1, cstride=1, linewidth
    =0.25,
55                          cmap=pylab.cm.jet, norm=matplotlib.colors.
    LogNorm())
56 axis.contour(x, y, s.T, levels, zdir='z', offset=-1.0,
57              norm=matplotlib.colors.LogNorm())
58 axis.contourf(x, y, s.T, 1, zdir='x', offset=-L)
59 axis.contourf(x, y, s.T, 1, zdir='y', offset=L)
60 axis.set_zlim(-1.0, 1.0)
61 axis.set_xlabel('x')
62 axis.set_ylabel('y')
63 axis.set_zlabel('u')
64 fig.colorbar(surf)
65 pylab.show()
```

Exercise 3.4 Solve the two-dimensional Laplace equation for the electric potential for a parallel plate capacitor with potentials $V = \pm 1\,\mathrm{V}$ contained within a grounded square region of side length L as shown in Fig. 3.18.

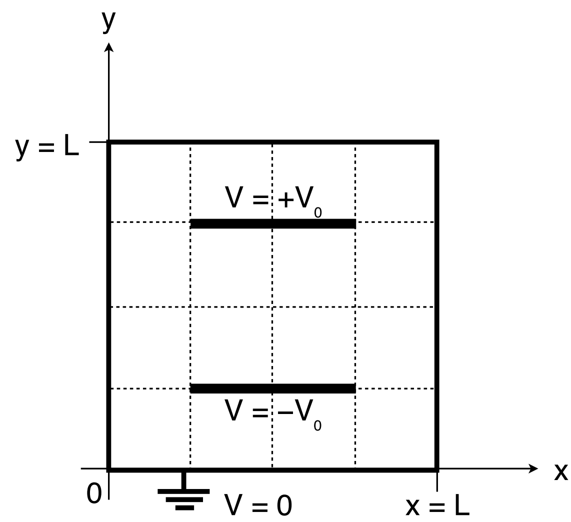

Figure 3.18: Example boundary value problem for a two-dimensional capacitor contained in a square region with sides of length L kept at 0 V.

Chapter 4

Random systems

4.1 Random walks

To determine the statistical properties of many body systems, we often perform simulations of stochastic models of these systems. For example, if we put a drop of ink in a pool of water, the ink molecules slowly diffuse throughout the pool. The molecules move in a way that might be deterministic if we considered all of the collisions with the water molecules, but is well modeled as a *random walk* in which each step is taken in a random direction. Generally, numerical methods form simulation of complex system via random numbers are known as *Monte Carlo methods*.

We consider first a one-dimensional random walk. Suppose a particle starts at position x_0. On each step the particle might move one unit in the positive- or negative-directions with equal probability. Thus, on step n, we have $x_n = x_{n-1} + \Delta x_n$ where the probability that $\Delta x_n = \pm\Delta x$ is positive is $1/2$ and the probability that it is negative is also $1/2$. Here, Δx is the step unit.

As time increases, the position of the particle will tend to drift away from zero, though it might tend to move off in the positive direction or in the negative direction. By step n, a particle will be at position

$$x_n = \sum_{i=1}^{n} \Delta x_i \tag{4.1}$$

though because the individual steps are random we cannot know where a particular particle will end up. We can make statistical statements about an ensemble of particles however. The expected value of the position of the random walk is zero, $\langle x_n \rangle$, since it is equally likely for the particle to wander off in the positive direction as it is to wander of in the negative direction. The *root-mean-squared distance* or RMS distance the typical particle will move by step n is

$$\sqrt{\langle x_n^2 \rangle} = \sqrt{\left\langle \left(\sum_{i=1}^{n} \Delta x_i \right)^2 \right\rangle} = \sqrt{\sum_{i=1}^{n} \sum_{j=1}^{n} \langle \Delta x_i \Delta x_j \rangle} = \sqrt{n}\,\Delta x \tag{4.2}$$

where we have used the fact that each step is independent so that $\langle \Delta x_i \Delta x_j \rangle = \delta_{ij}(\Delta x)^2$. Two random walkers will tend to diverge with time, but as the square-root of the step number.

If each step corresponds to a time Δt so that time t corresponds to step $n = t/\Delta t$ then we have

$$\langle x^2(t) \rangle = 2Dt \tag{4.3}$$

where $D = (\Delta x)^2/(2\Delta t)$ is the *diffusion constant*. The connection of random walks to diffusion can be seen as follows: suppose that there are a large number of particles undergoing independent random walks. We divide up our spatial domain into bins of some small but finite size and interpret the number of particles in each bin divided by the size of the bin as the number density in that bin. The process of binning up space into such bins is known as *coarse graining*. The number density as a function of time will be proportional to the probability of a given random walk finding itself in a bin as a function of time in an ensemble average (i.e., for a large number of particles) so we compute this probability.

For a single particle in a one-dimensional random walk, the probability of a particle finding itself at site $j\,\Delta x$ at time $(n+1)\Delta t$ depends only on the previous step at time $n\,\Delta t$: if the particle is at position $(j-1)\Delta x$ or at position $(j+1)\Delta x$ at time $n\,\Delta t$ then there is a probability of $1/2$ of the particle moving to position $j\,\Delta x$ at time $(n+1)\Delta t$. Thus,

$$\overset{n+1}{p}_j = \frac{1}{2}\,\overset{n}{p}_{j-1} + \frac{1}{2}\,\overset{n}{p}_{j+1} \tag{4.4}$$

which can be rewritten as

$$\frac{\overset{n+1}{p}_j - \overset{n}{p}_j}{\Delta t} = \frac{(\Delta x)^2}{2\Delta t}\,\frac{\overset{n}{p}_{j+1} - 2\,\overset{n}{p}_j + \overset{n}{p}_{j-1}}{(\Delta x)^2}. \tag{4.5}$$

This is simply a finite difference equation (the FTCS scheme) corresponding to the diffusion equation

$$\frac{\partial p(t,x)}{\partial t} = D\,\frac{\partial^2 p(t,x)}{\partial x^2} \tag{4.6}$$

with diffusion constant $D = (\Delta x)^2/(2\Delta t)$.

A numerical simulation of a random walk is given in Listing 4.1. This program computes several random walks. With each step of a given walk, a random value is chosen that determines whether to step forward or backward. In addition, the root-mean-squared distance moved averaged over all the walks is computed for each step. In Fig. 4.1 we show three sample random walks and in Fig. 4.2 we show the root-mean-squared distance traveled as a function of the step number. The latter figure shows that the expected behavior $x_{\mathrm{rms}} = \sqrt{n}$ where n is the step number is observed.

Listing 4.1: Program `randwalk.py`

```
import pylab, random

```

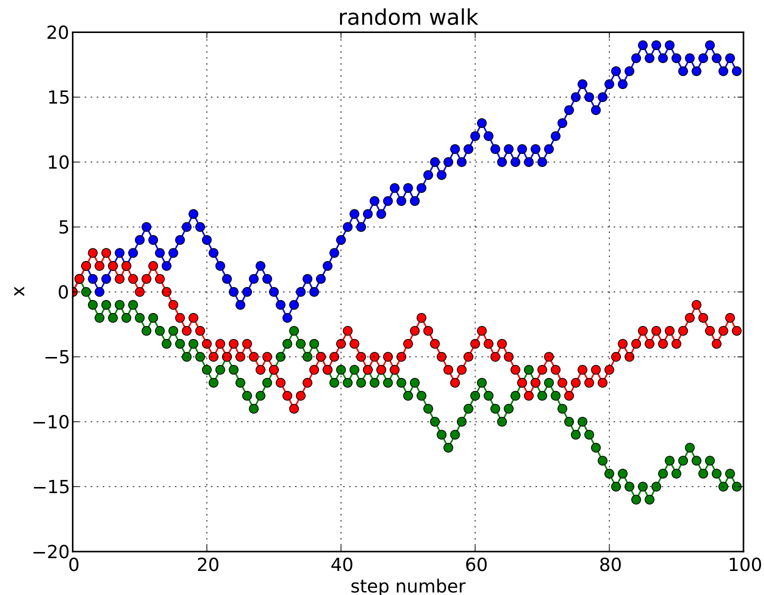

Figure 4.1: Results from running the program `randwalk.py` (Listing 4.1) for three random walks of 100 steps. The position of the walker with each step is shown.

```
3  nsteps = input('number of steps in walk -> ')
4  nwalks = input('number of random walks -> ')
5  seed = input('random number seed -> ')
6  random.seed(seed)
7  steps = range(nsteps)
8  xrms = [0.0]*nsteps   # mean squared distance
9
10 # loop over the number of walks being done
11 for i in range(nwalks):
12     x = [0]*nsteps   # position at each step in walk
13     # loop over steps in this walk
14     for n in steps[1:]:
15         x[n] = x[n-1]+random.choice([-1, +1])
16         xrms[n] += (x[n]**2-xrms[n])/(i+1)
17     pylab.plot(steps, x, 'o-')
18 for n in steps:
19     xrms[n] = xrms[n]**0.5
20
21 pylab.title('random walk')
22 pylab.xlabel('step number')
23 pylab.ylabel('x')
```

Figure 4.2: Results from running the program `randwalk.py` (Listing 4.1) for 500 random walks of 100 steps. The root-mean-squared distance from the origin of the walks at each step is shown. The solid line is the predicted $x_{\mathrm{rms}} = \sqrt{n}$ where n is the step number.

```
24 pylab.grid()
25 pylab.figure()
26 pylab.title('root-mean-squared distance for %d walks'%nwalks)
27 pylab.plot(steps, xrms, '.')
28 pylab.plot(steps, [n**0.5 for n in steps], '-')
29 pylab.xlabel('step number')
30 pylab.ylabel('root-mean-squared distance')
31 pylab.grid()
32 pylab.show()
```

Now consider a random walk in two dimensions: a number of particles are arranged in a square lattice in the middle of a square container. Each particle then undergoes independent random walks. The initially ordered arrangement diffuses to fill the entire box. This could be a model of, for example, a drop of ink placed in a pool of water.

The system will naturally evolve from an ordered state to a disordered state. To get some insight into why this occurs, suppose that we divide the square region into some number K subregions or bins so that there are a large number of bins, $K \gg 1$, but K is much smaller than the number of particles, $K \ll N$, and there are

a large number of sites within each bin. This is essentially the coarse-graining that we discussed earlier. A *state* is a particular arrangement the N particles amongst the K bins, while the *distribution* is given by the set of *occupation numbers*, $\{n_k\}$ for $k = 0, 1, \dots, K-1$, that describe the number of particles in each bin. We assume that, in equilibrium, each state is equally likely, but some distributions (sets of occupation numbers) will be more likely than others. The reason that some sets of occupation numbers are more likely than others is because some sets of occupation numbers will more ways of being realized than others. The *multiplicity of states* describes the number of ways of realizing a particular distribution; it is given by

$$\Omega_{\{n_k\}} = \frac{N!}{n_0!n_1!\cdots n_{K-1}!}. \tag{4.7}$$

If each state is equally likely then the distribution with the largest value of the multiplicity of states will be the most likely distribution.

To find the most likely distribution, then, we need to maximize $\Omega_{\{n_k\}}$ over all possible sets of occupation numbers, $\{n_k\}$. It is easier to consider the natural logarithm of $\Omega_{\{n_k\}}$, so define the *entropy*

$$S_{\{n_k\}}/k_\mathrm{B} = \ln \Omega_{\{n_k]\}} = \ln N! - \sum_{k=0}^{K-1} \ln n_k!. \tag{4.8}$$

where

$$k_\mathrm{B} = 1.3806503 \times 10^{-23} \mathrm{m}^2\,\mathrm{kg}\,\mathrm{s}^{-2}\,\mathrm{K}^{-1} \tag{4.9}$$

is Boltzmann's constant. Since $S_{\{n_k\}}$ is monotonic in $\Omega_{\{n_k\}}$, the distribution that maximizes $\Omega_{\{n_k\}}$ will also maximize $S_{\{n_k\}}$. Consider a small perturbation to the occupation numbers, $\{\delta n_k\}$. At the maximum point we have

$$\begin{aligned} 0 = (S_{\{n_k+\delta n_k\}} - S_{\{n_k\}})/k_\mathrm{B} &= -\sum_{k=0}^{K-1} \ln\left(\frac{(n_k+\delta n_k)!}{n_k!}\right) \\ &= -\sum_{k=0}^{K-1} \ln[(n_k+1)(n_k+2)\cdots(n_k+\delta n_k)] \\ &\approx -\sum_{k=0}^{K-1} \delta n_k \ln n_k \end{aligned} \tag{4.10}$$

where we assume that $\delta n_k \ll n_k$ so that each factor in the logarithm is $\approx n_k$. Therefore we want to find the values $\{n_k\}$ for which

$$\delta n_0 \ln n_0 + \delta n_1 \ln n_1 + \delta n_2 \ln n_2 + \cdots + \delta n_{K-1} \ln n_{K-1} = 0 \tag{4.11}$$

for arbitrary $\{\delta n_k\}$ which would result in $n_0 = n_1 = \cdots = n_{K-1} = 1$. However this cannot be the right answer because then there would only be K particles, not N particles. We need to enforce the constraint

$$\sum_{k=0}^{K-1} n_k = N = \text{constant} \tag{4.12}$$

which results in

$$\delta n_0 + \delta n_1 + \delta n_2 + \cdots + \delta n_{K-1} = 0 \tag{4.13}$$

or $\delta n_0 = -\delta n_1 - \delta n_2 - \cdots - \delta n_{K-1}$. We substitute this into Eq. (4.11) to obtain

$$(\ln n_1 - \ln n_0)\delta n_1 + (\ln n_2 - \ln n_0)\delta n_2 + \cdots + (\ln n_{K-1} - \ln n_0)\delta n_{K-1} = 0. \tag{4.14}$$

This equation will hold if $n_0 = n_1 = n_2 = \cdots = n_{K-1} = N/K$, i.e., when all the bins have the same number of particles.

Incidentally, it is instructive to consider the situation in particles in different bins have different energies, e.g., if we consider diffusion of particles in a vertical plane in which particles prefer to move downward rather than upward. Let us suppose that each particle in the kth bin has energy ϵ_k. The total energy E is conserved and so we have an additional constraint that

$$\sum_{k=1}^{K-1} \epsilon_k n_k = E = \text{constant} \tag{4.15}$$

or

$$\epsilon_1 \delta n_1 + \epsilon_2 \delta n_2 + \cdots + \epsilon_{K-1} \delta n_{K-1} = 0 \tag{4.16}$$

where, for convenience, we have taken the zero of energy such that $\epsilon_0 = 0$. We substitute this additional constraint, $\delta n_1 = -(\epsilon_2/\epsilon_1)\delta n_2 - \cdots - (\epsilon_2/\epsilon_{K-1})\delta n_{K-1}$, into Eq. (4.14) and we find

$$(\ln n_2 - \ln n_0 + \beta\epsilon_2)\delta n_2 + \cdots + (\ln n_{K-1} - \ln n_0 + \beta\epsilon_{K-1})\delta n_{K-1} = 0 \tag{4.17}$$

where

$$\beta = -\frac{\ln(n_1/n_0)}{\epsilon_1} \tag{4.18}$$

is a Lagrange multiplier which can be understood to be the reciprocal of the temperature of the system,

$$\beta = \frac{1}{k_\mathrm{B} T}. \tag{4.19}$$

Now we find that the most likely set of occupation numbers are those for which

$$n_k = n_0 \mathrm{e}^{-\beta\epsilon_k}. \tag{4.20}$$

Using the constraint equations for E and N we can determine the constants n_0 and β. If we define the *partition function*,

$$Z(\beta) = \sum_{k=0}^{K-1} \mathrm{e}^{-\beta\epsilon_k} \tag{4.21}$$

then we see that

$$N = \sum_{k=0}^{K-1} n_k = \sum_{k=0}^{K-1} n_0 \mathrm{e}^{-\beta\epsilon_k} = n_0 Z(\beta) \tag{4.22}$$

so

$$n_0 = \frac{N}{Z(\beta)} \quad \text{and} \quad n_k = \frac{N}{Z(\beta)}\, \mathrm{e}^{-\beta\epsilon_k} \tag{4.23}$$

or, in terms of the probability that a particle will be in bin k, $P_k = n_k/N$,

$$P_k = \frac{1}{Z(\beta)}\, \mathrm{e}^{-\beta\epsilon_k} \tag{4.24}$$

Furthermore,

$$E = \sum_{k=0}^{K-1} n_k \epsilon_k = \frac{N}{Z(\beta)} \sum_{k=0}^{K-1} \epsilon_k \mathrm{e}^{-\beta\epsilon_k} = -N \frac{d}{d\beta} \ln Z(\beta) \tag{4.25}$$

which will determine β in terms of E and N.

We can use *Stirling's approximation*, $\ln n! \sim n \ln n - n$ for large n, to express the logarithm of the multiplicity of states in a more convenient form:

$$\begin{aligned} S/k_\mathrm{B} = \ln \Omega = \ln N! - \sum_{k=0}^{K-1} \ln n_k! \\ &\approx (N \ln N - N) - \sum_{k=0}^{K-1} (n_k \ln n_k - n_k) \\ &= \sum_{k=0}^{K-1} (n_k \ln N - n_k \ln n_k) \\ &= -N \sum_{k=0}^{K-1} P_k \ln P_k \end{aligned} \tag{4.26}$$

where $P_k = n_k/N$ is the probability of any particular particle being in bin k. For the situation in which there is a different energy to occupy each bin we have

$$\begin{aligned} S/k_\mathrm{B} &\approx (N \ln N - N) - \sum_{k=0}^{K-1} (n_k \ln n_k - n_k) \\ &= N \ln N - \sum_{k=0}^{K-1} n_k \ln\left(\frac{N}{Z(\beta)} \mathrm{e}^{-\beta\epsilon_k} \right) \\ &= N \ln N - \sum_{k=0}^{K-1} \{ n_k \ln N - n_k \ln Z(\beta) - \beta n_k \epsilon_k \} \\ &= N \ln Z(\beta) + \beta E \end{aligned} \tag{4.27}$$

Now, for a small alteration of the energies associated with the bins, we have

$$dS = k_\mathrm{B} \left(N \frac{\partial \ln Z(\beta)}{\partial \beta} + E \right) d\beta + k_\mathrm{B} \beta\, dE = k_\mathrm{B} \beta\, dE \tag{4.28}$$

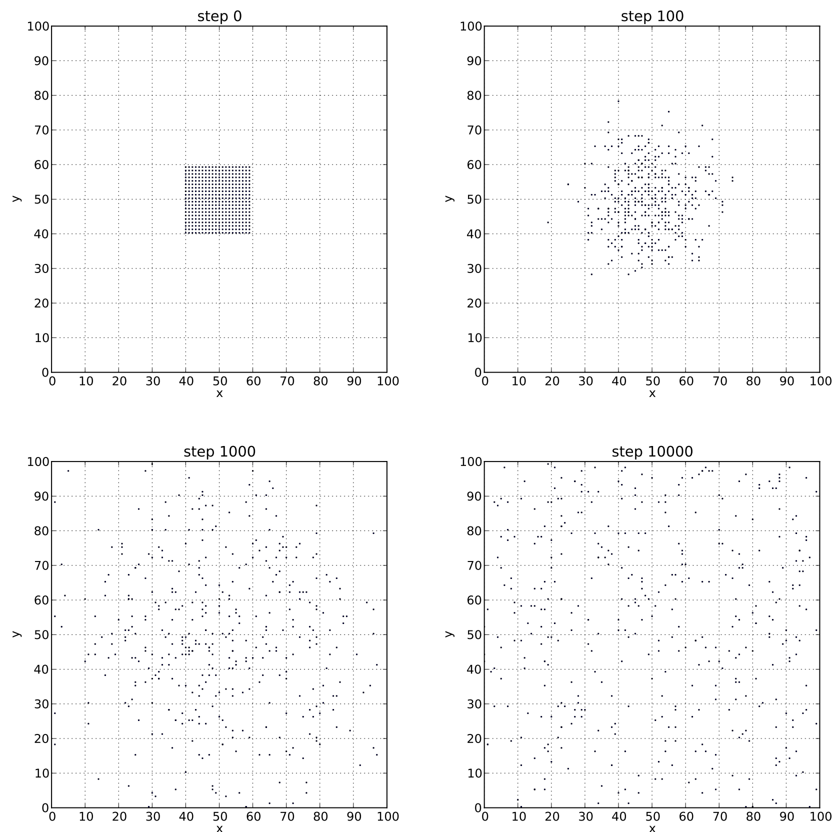

Figure 4.3: Snapshots from running the program `diffuse.py` (Listing 4.2) at four times. The distribution of particles is initially arranged in a 20×20 square in the middle of a 100×100 box. With time the particles diffuse until they fill the box homogeneously.

which can be written in the familiar form

$$dE = T\,dS. \tag{4.29}$$

For our simulation of diffusion of particles in two-dimensions, the particles are initially arranged in a square of dimensions $M \times M$ in the middle of a square box of dimensions $L \times L$. To monitor the growth of entropy, the box is divided into $K = B \times B$ square bins. As the simulation progresses, fraction of the particles in each bin is used to compute P_k, and this is then used to compute the entropy per particle at each time step (in units of Boltzmann's constant). The program to perform this simulation is giving in Listing 4.2. Snapshots of the evolution taken at several times are shown in Fig. 4.3 and a plot showing the increase of entropy with time is shown in Fig. 4.4.

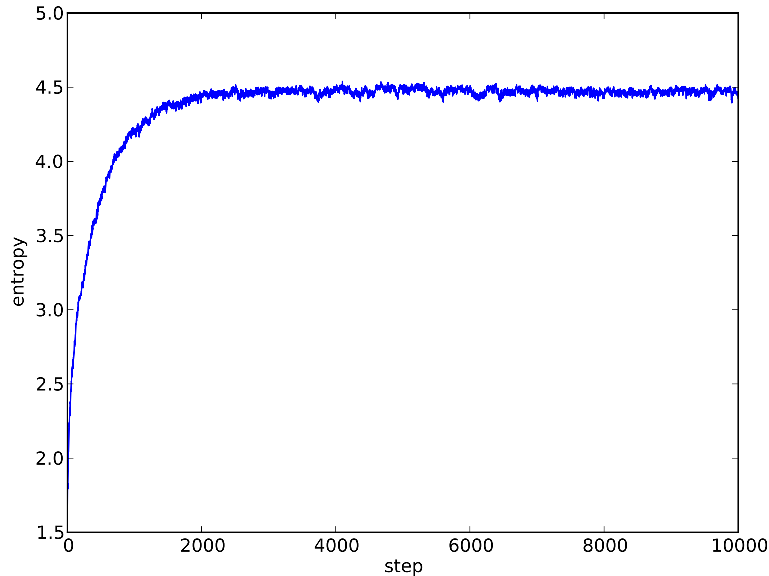

Figure 4.4: Entropy evolution from the program `diffuse.py` (Listing 4.2) for the evolution shown in Fig. 4.3. As the initially highly-ordered system diffuses throughout the box, the entropy increases.

Listing 4.2: Program `diffuse.py`

```
1  import pylab, math, random
2
3  L = 100  # length of side of box
4  M = 20  # length of side of square of particles
5  B = 10  # number of coarse−grainig bins per side
6  nsteps = input('number of steps in walk -> ')
7  steps = range(nsteps)
8  seed = input('random number seed -> ')
9  random.seed(seed)
10
11 # initial positions of particles form a MM block in the middle of the box
12 xside = range((L−M)//2, (L+M)//2)
13 yside = range((L−M)//2, (L+M)//2)
14 x = [i for i in xside for j in yside]  # x−locations of the particles
15 y = [j for i in xside for j in yside]  # y−locations of the particles
16 N = len(xside)*len(yside)  # number of particles
17 S = [0.0]*nsteps  # entropy
18 P = pylab.zeros((B, B))  # probability of particle being in each bin
19
20 # setup animated figure
```

```
21 pylab.figure(figsize=(6, 6))
22 (points, ) = pylab.plot(x, y, ',')
23 pylab.xlim(0, L)
24 pylab.ylim(0, L)
25 pylab.xticks(range(0, L+1, L//B))
26 pylab.yticks(range(0, L+1, L//B))
27 pylab.xlabel('x')
28 pylab.ylabel('y')
29 pylab.grid()
30
31 # simulate the random walks
32 for n in steps:
33     # update plot
34     points.set_data(x, y)
35     pylab.title('step %d'%n)
36     pylab.pause(1e-6)
37     pylab.draw()
38
39     # update positions of particles and update counts in bins
40     P.fill(0)
41     for i in range(N):
42         (dx, dy) = random.choice([(-1, 0), (1, 0), (0, -1), (0, 1)])
43         x[i] += dx
44         y[i] += dy
45         # make sure that the particles stay in the box
46         if x[i] < 0 or x[i] >= 100:
47             x[i] -= dx
48         if y[i] < 0 or y[i] >= 100:
49             y[i] -= dy
50         # increment count in bin containing particle
51         P[x[i]*B//L,y[i]*B//L] += 1.0
52
53     # compute the entropy at this step
54     for i in range(B):
55         for j in range(B):
56             P[i,j] /= N
57             if P[i,j] > 0:
58                 S[n] -= P[i,j]*math.log(P[i,j])
59 pylab.figure()
60 pylab.plot(steps, S)
61 pylab.xlabel('step')
62 pylab.ylabel('entropy')
63 pylab.show()
```

Exercise 4.1 Consider a vertical column of height h containing gas molecules. Initially, the molecules are contained in a small region in the middle of the column, but then they diffuse to fill the entire column. Because of gravity, the density of molecules will be slightly greater near the base of the column than it will be at the top. Simulate this system as a random walk of some number N of molecules but where the probability, a, to increase one step in height, $+\Delta y$, is slightly different from the probability, $1-a$, to decrease one step in height, $-\Delta y$. Derive an expression for a given in terms of the molecular mass M of the gas, the standard freefall $g = 9.8\,\mathrm{m\,s^{-2}}$, the temperature of the gas, T, the vertical height step, Δy, the Boltzmann conststant k_{B} and the atomic mass unit $\mathrm{u} = 1.660\,538\,921 \times 10^{-27}\,\mathrm{kg}$. Perform a numerical simulation and show that the expected distribution of particles with height is obtained after the gas reaches equilibrium (once the entropy becomes approximately constant with time). Try the values $T = 273\,\mathrm{K}$, $\Delta y = 100\,\mathrm{m}$, $h = 10\,\mathrm{km}$, and $M = 28.964\,\mathrm{u}$ for dry air.

4.2 Ising model

The *Ising model* is a model of a ferromagnetic material. The material consists of a large number of atoms arranged in a lattice. Each atom has two states of spin, spin-up or spin-down, and these spins interact with each other via their mutual magnetic interaction and they also interact with any ambient magnetic field. At high temperature the spins all have random directions (either up or down) but as the material is cooled the magnetic interactions tend to align the spins.

Let $\{s_i\}$ for $i = 0, 1, 2, \ldots N-1$ be the values of the spins where each spin can have a value of either $+1$ (spin up) or -1 (spin down). Only nearest neighbor interactions are considered in the Ising model; the energy of the system is thus computed in terms of *pairs* of nearest atoms, $\langle i, j\rangle$, as

$$E = -J \sum_{\langle i,j\rangle} s_i s_j - \mu H \sum_{i=0}^{N-1} s_i \tag{4.30}$$

where J is the *exchange energy*, H is an externally applied magnetic field, which we will take to be $H = 0$ at first, and μ is the magnetic moment of the spins. The overall magnetization is given by

$$M = \sum_{i=0}^{N-1} s_i . \tag{4.31}$$

As mentioned above, at high temperatures, the spins fluctuate randomly and $\langle M\rangle = 0$; the material is said to be in a *paramagnetic phase*. As the temperature drops below a critical temperature known as the *Curie temperature* T_{c}, there is a phase transition

in which the spins begin align and the material becomes magnetized. This is known as the *ferromagnetic phase*.

To get a sense of the nature of the phase transition, it is helpful to analyze the problem under an approximation known as *mean field theory*. At any particular temperature the expectation value of any of the spins is $\langle s \rangle$. The expected value of the magnetization is therefore

$$\langle M \rangle = N \langle s \rangle . \tag{4.32}$$

The energy of the system can be written in the form

$$E = \sum_i s_i \left\{ -J \sum_{j \in \langle i,j \rangle} s_j - \mu H \right\} \tag{4.33}$$

which indicates that we can regard atom i to experience an effective magnetic field

$$H_{\text{eff}} = H + \frac{J}{\mu} \sum_{j \in \langle i,j \rangle} s_j \tag{4.34}$$

which contains both the applied magnetic field H and the field created by the neighboring particles. In the mean field theory we have

$$\langle H_{\text{eff}} \rangle = \frac{zJ}{\mu} \langle s \rangle \tag{4.35}$$

where we take the applied field to vanish, $H = 0$, and where z is the number of neighbors to each atoms. In one-dimension, $z = 2$; in a two two-dimensional rectilinear grid, $z = 4$; in a three-dimensional rectilinear grid, $z = 6$. The contribution to the energy of spin i in this mean effective field is $E_\pm = \mp \mu \langle H_{\text{eff}} \rangle$ where E_+ is the energy if $s_i = +1$ and E_- is the energy if $s_i = -1$. In thermal equilibrium the relative probabilities of the two states are

$$\frac{P_+}{P_-} = \mathrm{e}^{-(E_+ - E_-)/k_{\mathrm{B}} T} = \mathrm{e}^{2\mu \langle H_{\text{eff}} \rangle / k_{\mathrm{B}} T} \tag{4.36}$$

and so

$$P_\pm = \frac{\mathrm{e}^{\pm \mu \langle H_{\text{eff}} \rangle / k_{\mathrm{B}} T}}{\mathrm{e}^{+\mu \langle H_{\text{eff}} \rangle / k_{\mathrm{B}} T} + \mathrm{e}^{-\mu \langle H_{\text{eff}} \rangle / k_{\mathrm{B}} T}} . \tag{4.37}$$

Now we can compute the expectation value of s_i sitting in the mean effective field:

$$\begin{aligned} \langle s_i \rangle &= \sum_{s_i \in \{+,-\}} s_i P_{s_i} = P_+ - P_- = \frac{\mathrm{e}^{+\mu \langle H_{\text{eff}} \rangle / k_{\mathrm{B}} T} - \mathrm{e}^{-\mu \langle H_{\text{eff}} \rangle / k_{\mathrm{B}} T}}{\mathrm{e}^{+\mu \langle H_{\text{eff}} \rangle / k_{\mathrm{B}} T} + \mathrm{e}^{-\mu \langle H_{\text{eff}} \rangle / k_{\mathrm{B}} T}} \\ &= \tanh (\mu \langle H_{\text{eff}} \rangle / k_{\mathrm{B}} T) . \end{aligned} \tag{4.38}$$

Because all atoms are equivalent, $\langle s_i \rangle = \langle s \rangle$ so from Eq. (4.35) and Eq. (4.38) we have

$$\langle s \rangle = \tanh (zJ \langle s \rangle / k_{\mathrm{B}} T) . \tag{4.39}$$

From this equation we can compute the value of $\langle s\rangle$ for any temperature. One solution is always $\langle s\rangle = 0$, but at low temperature there are three solutions, and the $\langle s\rangle = 0$ solution is not the lowest energy one. There is a critical temperature at which this transition occurs, which we can understand by expanding the hyperbolic tangent function in powers of its argument near the critical temperature where $\langle s\rangle \approx 0$. Since $\tanh x = x - \frac{1}{3}x^3 + O(x^5)$ we have

$$\langle s\rangle \approx \frac{zJ\langle s\rangle}{k_\mathrm{B}T} - \frac{1}{3}\left(\frac{zJ\langle s\rangle}{k_\mathrm{B}T}\right)^3 \tag{4.40}$$

which can be solved for $\langle s\rangle$ to obtain

$$\begin{aligned}\langle s\rangle &\approx \frac{k_\mathrm{B}T}{zJ}\sqrt{\frac{3k_\mathrm{B}}{zJ}\left(\frac{zJ}{k_\mathrm{B}} - T\right)} \qquad \text{for } T \le T_\mathrm{c} \\ &\sim (T_\mathrm{c} - T)^{1/2} \qquad \text{for } T \lesssim T_\mathrm{c}.\end{aligned} \tag{4.41}$$

where $T_\mathrm{c} = zJ/k_\mathrm{B}$. We see therefore that there is an abrupt transition during a cooling process from the paramagnetic state to the ferromagnetic state at the Curie temperature T_c, and that point the magnetization behaves as a power-law with a *critical exponent* which in the mean field theory is $1/2$. Note that although the magnetization is not a smooth function of temperature at the Curie temperature, it is continuous. This is the characteristic of a *second-order phase transition*. For a two-dimensional lattice with $z = 4$, the mean field theory predicts a critical temperature $T_\mathrm{c} = 4J/k_\mathrm{B}$.

An exact solution is known (Onsanger 1944). The critical temperature is

$$T_\mathrm{c} = \frac{2J/k_\mathrm{B}}{\ln(1+\sqrt{2})} \approx 2.269\,\frac{J}{k_\mathrm{B}} \tag{4.42}$$

and the expected value $\langle s\rangle$ is

$$\begin{aligned}\langle s\rangle &= \left[1 - \sinh^{-4}(2J/k_\mathrm{B}T)\right]^{1/8} \qquad \text{for } T \le T_\mathrm{c} \\ &\sim (T_\mathrm{c} - T)^{1/8} \qquad \text{for } T \lesssim T_\mathrm{c}.\end{aligned} \tag{4.43}$$

Although the critical temperature and the critical exponent are different from those predicted by the mean field theory, we see that the magnetization is again continuous but not smooth function of temperature at the critical temperature.

In addition to the energy and magnetization, we can compute other thermodynamic variables during the evolution, such as the heat capacity and the magnetic susceptability. From the definition of the partition function Z we can obtain a few important relations:

$$\frac{\partial Z}{\partial \beta} = \frac{\partial}{\partial \beta}\sum_{i=0}^{N-1} \mathrm{e}^{-\beta E_i} = -\sum_{i=0}^{N-1} E_i \mathrm{e}^{-\beta E_i} = -Z\sum_{i=0}^{N-1} E_i P_i = -Z\langle E\rangle \tag{4.44}$$

$$\frac{\partial^2 Z}{\partial \beta^2} = \frac{\partial^2}{\partial \beta^2}\sum_{i=0}^{N-1} \mathrm{e}^{-\beta E_i} = \sum_{i=0}^{N-1} E_i^2 \mathrm{e}^{-\beta E_i} = Z\sum_{i=0}^{N-1} E_i^2 P_i = Z\langle E^2\rangle \tag{4.45}$$

where E_i is the energy of atom i and $P_i = Z^{-1}\mathrm{e}^{-\beta E_i}$ is the probability of the atom i to have this energy. We then obtain a useful formula for the *heat capacity* of the system,

$$\begin{aligned} C = \frac{\partial \langle E\rangle}{\partial T} &= -\frac{1}{k_\mathrm{B}T^2}\frac{\partial \langle E\rangle}{\partial \beta} = \frac{1}{k_\mathrm{B}T^2}\frac{\partial}{\partial\beta}\left(\frac{1}{Z}\frac{\partial Z}{\partial \beta}\right) \\ &= \frac{1}{k_\mathrm{B}T^2}\left[\frac{1}{Z}\frac{\partial^2 Z}{\partial\beta^2} - \left(\frac{1}{Z}\frac{\partial Z}{\partial\beta}\right)^2\right] \\ &= \frac{1}{k_\mathrm{B}T^2}\,\mathrm{Var}(E) \end{aligned} \tag{4.46}$$

where $\mathrm{Var}(E) = \langle(\Delta E)^2\rangle = \langle E^2\rangle - \langle E\rangle^2$ is the variance of the fluctuations in the energy. Thus we can compute the heat capacity of the system by measuring the fluctuation of the energy at fixed temperature rather than by the direct approach of measuring the change of energy with temperature.

A similar trick can be done by taking derivatives with respect to the externally-applied magnetic field H:

$$\frac{\partial Z}{\partial H} = \frac{\partial}{\partial H}\sum_{i=0}^{N-1}\mathrm{e}^{-\beta E_i} = \sum_{i=0}^{N-1}\beta\mu s_i \mathrm{e}^{-\beta E_i} = \beta\mu Z\sum_{i=0}^{N-1} s_i P_i = \beta\mu Z\langle M\rangle \tag{4.47}$$

$$\frac{\partial^2 Z}{\partial H^2} = \frac{\partial^2}{\partial H^2}\sum_{i=0}^{N-1}\mathrm{e}^{-\beta E_i} = \sum_{i=0}^{N-1}\beta^2\mu^2 s_i^2 \mathrm{e}^{-\beta E_i} = \beta^2\mu^2 Z\sum_{i=0}^{N-1} s_i^2 P_i = \beta^2\mu^2 Z\langle M^2\rangle \tag{4.48}$$

which yield an expression for the *magnetic susceptibility*

$$\begin{aligned} \chi = \frac{\partial\langle M\rangle}{\partial H} &= \frac{k_\mathrm{B}T}{\mu}\frac{\partial}{\partial H}\left(\frac{1}{Z}\frac{\partial Z}{\partial H}\right) \\ &= \frac{k_\mathrm{B}T}{\mu}\left[\frac{1}{Z}\frac{\partial^2 Z}{\partial H^2} - \left(\frac{1}{Z}\frac{\partial Z}{\partial H}\right)^2\right] \\ &= \frac{\mu}{k_\mathrm{B}T}\,\mathrm{Var}(M). \end{aligned} \tag{4.49}$$

Note that we can compute the magnetic susceptibility of the system even when there is no externally applied magnetic field.

We now turn to the numerical simulation of the Ising model. We start with a two-dimensional lattice of $L\times L$ atoms. Initially, the system is cold with all spins are aligned with $s_i = +1$ for all i, and the temperature is kept at a small value. We now perform a number of sweeps over all of the atoms where we perform the following procedure: for each atom we compute the energy required to flip a spin, E_flip. If this energy is less than zero, then we flip the spin, but even when there is a energy cost to such a flip, we will allow the transition with a probability

$$P_\mathrm{flip} = \mathrm{e}^{-E_\mathrm{flip}/k_\mathrm{B}T}. \tag{4.50}$$

This is known as the *Metropolis algorithm*. Thus, at non-zero temperature, there is a chance that an atom will undergo a transition to a higher energy state. The average

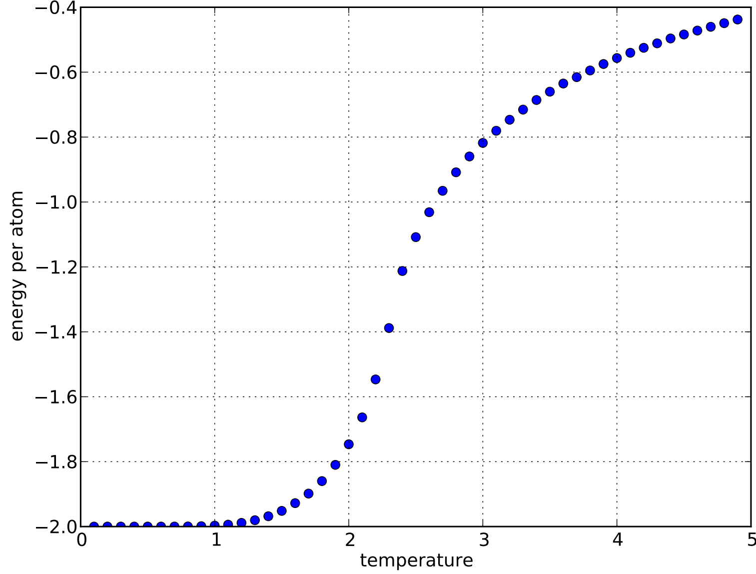

Figure 4.5: The energy per atom as a function of temperature for the Ising model obtained from the program `ising.py` (Listing 4.3). These results were obtained for a 30×30 lattice of atoms with an average of 10^4 sweeps for each temperature.

energy and magnetization of the system over many sweeps through the atoms in the lattice for fixed temperature, and then the temperature is increased slightly and the procedure is repeated.

To see that the Metropolis algorithm for allowing a transition that increases the energy with a probability given by Eq. (4.50) yields the correct distribution, note that this procedure implies that the rate of transition from state 1 to state 2 where $E_1 > E_2$ is $W(1 \to 2) = 1$ while the rate of transitions from state 2 to state 1 is $W(2 \to 1) = \exp(-E_{\text{flip}}/k_{\text{B}}T)$ where $E_{\text{flip}} = E_1 - E_2$. In equilibrium we require the number of transitions $1 \to 2$ to be equal to the number of transitions $2 \to 1$, so we have

$$P_1 W(1 \to 2) = P_2 W(2 \to 1) \tag{4.51}$$

which implies

$$\frac{P_1}{P_2} = \mathrm{e}^{-(E_1 - E_2)/k_{\text{B}}T} \tag{4.52}$$

and this is the desired distribution.

Listing 4.3 presents a program that will simulate a two-dimensional lattice of $L \times L$ atoms with periodic boundary conditions. Initially, all the spins are aligned in the up-direction and the temperature is low, $T = 0.1$ where hereafter temperature

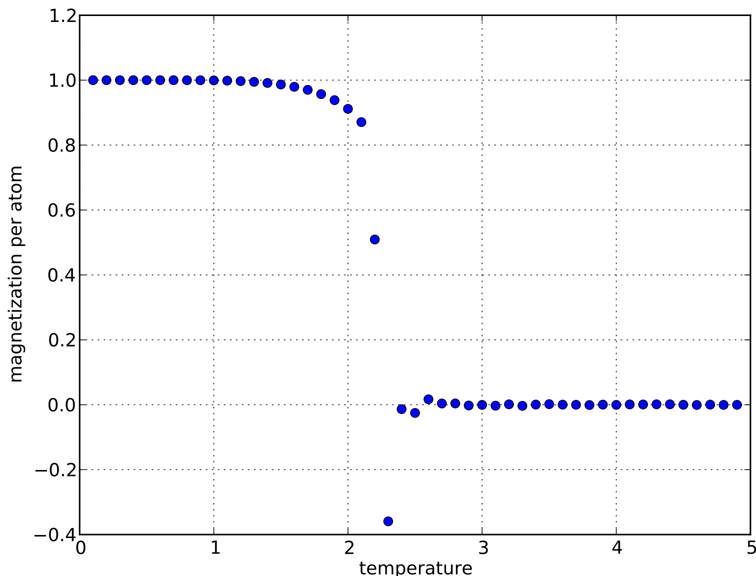

Figure 4.6: The magnetization per atom as a function of temperature for the Ising model obtained from the program `ising.py` (Listing 4.3). These results were obtained for a 30×30 lattice of atoms with an average of 10^4 sweeps for each temperature.

is measured in units of J/k_B. The temperature is incrementally increased in small steps while a large number of sweeps of the Metropolis algorithm over the lattice are performed at each temperature and the average energy per atom in units of J (Fig. 4.5), magnetization per atom in units of J/μ (Fig. 4.6), and heat capacity per atom in units of k_B (Fig. 4.7). Note that the magnetization vanishes for temperatures above the Curie temperature T_c, and at this temperature there is a cusp in the heat capacity (a change in energy of the system does not cause a change in temperature at this temperature).

Listing 4.3: Program `ising.py`

```
1  import math, pylab, random
2
3  J = 1.0  # exchange energy
4  L = input('number of atoms per side of lattice -> ')
5  nsweep = input('number sweeps to average -> ')
6  seed = input('random number seed -> ')
7  random.seed(seed)
8  N = L**2
9  kT = pylab.arange(0.1, 5.0, 0.1)
10 e = pylab.zeros(len(kT))
```

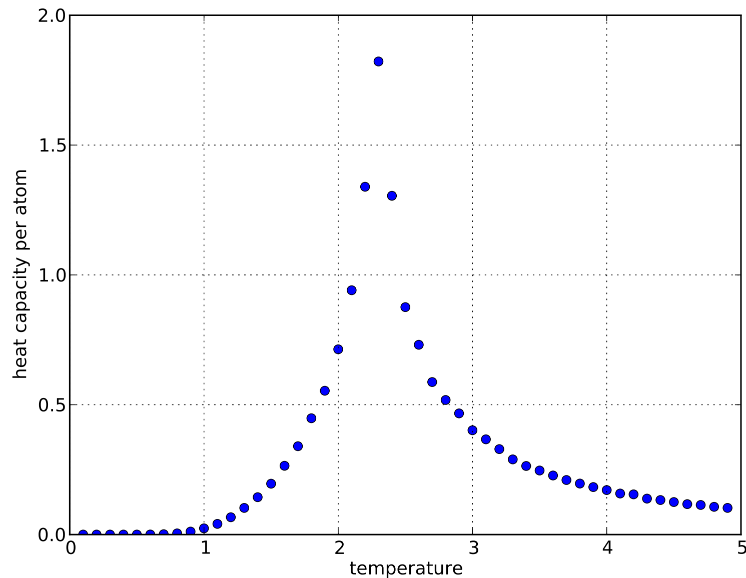

Figure 4.7: The heat capacity per atom as a function of temperature for the Ising model obtained from the program `ising.py` (Listing 4.3). These results were obtained for a 30×30 lattice of atoms with an average of 10^4 sweeps for each temperature.

```
11 m = pylab.zeros(len(kT))
12 c = pylab.zeros(len(kT))
13
14 # initial data
15 s = pylab.ones((L, L))
16 E = 0.0
17 M = 0.0
18 for i in range(L):
19     for j in range(L):
20         E -= J*s[i,j]*(s[(i+1)%L,j]+s[i,(j+1)%L])
21         M += s[i,j]
22
23 # prepare animated plot
24 pylab.ion()
25 image = pylab.imshow(s, vmax=1, vmin=-1)
26
27 # slowly warm up
28 for t in range(len(kT)):
29     # average nsweep sweeps
30     for sweep in range(nsweep):
```

```
31
32          # update animated plot
33          image.set_data(s)
34          pylab.title('kT = %g'%kT[t])
35          pylab.draw()
36
37          # sweep over all particles in lattice
38          for i in range(L):
39              for j in range(L):
40
41                  # compute energy required to flip spin
42                  dE = s[(i+1)%L,j]+s[(i-1)%L,j]+s[i,(j+1)%L]+s[i,(j-1)%L
    ]
43                  dE *= 2.0*J*s[i,j]
44
45                  # Metropolis algorithm to see if we should accept trial
46                  if dE <= 0.0 or random.random() <= math.exp(-dE/kT[t]):
47                      # accept trial: reverse spin; return dE and dM
48                      s[i,j] *= -1
49                      M += 2.0*s[i,j]
50                      E += dE
51
52          # update running means and variances
53          deltae = E-e[t]
54          deltam = M-m[t]
55          e[t] += deltae/(sweep+1)
56          m[t] += deltam/(sweep+1)
57          c[t] += deltae*(E-e[t])
58      e[t] /= N
59      m[t] /= N
60      c[t] /= nsweep*N*kT[t]**2
61
62  # produce plots
63  pylab.ioff()
64  pylab.figure()
65  pylab.plot(kT, e, 'o')
66  pylab.xlabel('temperature')
67  pylab.ylabel('energy per atom')
68  pylab.grid()
69  pylab.figure()
70  pylab.plot(kT, m, 'o')
71  pylab.xlabel('temperature')
72  pylab.ylabel('magnetization per atom')
73  pylab.grid()
74  pylab.figure()
75  pylab.plot(kT, c, 'o')
76  pylab.xlabel('temperature')
77  pylab.ylabel('heat capacity per atom')
78  pylab.grid()
79  pylab.show()
```

Exercise 4.2 First-order phase transitions. A first-order phase transition is a transition in which some state variable is a discontinuous function of some intensive variable. Consider the Ising model with an externally-applied field H. Investigate how the magnetization of a two-dimensional lattice of spins behaves as a function of H at constant temperature for temperatures between $1J/k_\mathrm{B}$ and $5J/k_\mathrm{B}$. Notice that there is a first order phase transition (a discontinuous jump in M as a function of H) for temperatures below T_c.

4.3 Variational method

In quantum mechanics, we often want to find the ground state of some system with a complicated potential. The *variational method* uses the fact that the energy functional of trial wave functions ψ,

$$E[\psi] = \frac{\int \psi^*(\boldsymbol{x})\hat{H}\psi(\boldsymbol{x})\,d\boldsymbol{x}}{\int |\psi(\boldsymbol{x})|^2\,d\boldsymbol{x}}, \tag{4.53}$$

is a minimum for the ground state. This is easily seen by expressing the trial wave function ψ in terms of the eigenfunctions ψ_n, $n = 0, 1, 2, \ldots$, of the Hamiltonian, which satisfy $\hat{H}\psi_n = E_n\psi_n$ with $E_0 < E_1 < E_2 < \cdots$,

$$\psi = \sum_{n=0}^{\infty} c_n\psi_n. \tag{4.54}$$

Then we have

$$\int \psi^*(\boldsymbol{x})\psi(\boldsymbol{x})\,d\boldsymbol{x} = \sum_{n=0}^{\infty}\sum_{m=0}^{\infty} c_n^* c_m \int \psi_n^*(\boldsymbol{x})\psi_m(\boldsymbol{x})\,d\boldsymbol{x} = \sum_{n=0}^{\infty} |c_n|^2 \tag{4.55}$$

since $\int \psi_n^*(\boldsymbol{x})\psi_m^*(\boldsymbol{x})\,d\boldsymbol{x} = \delta_{mn}$, and

$$\begin{aligned}
\int \psi^*(\boldsymbol{x})\hat{H}\psi(\boldsymbol{x})\,d\boldsymbol{x} &= \sum_{n=0}^{\infty}\sum_{m=0}^{\infty} c_n^* c_m \int \psi_n^*(\boldsymbol{x})\hat{H}\psi_m(\boldsymbol{x})\,d\boldsymbol{x} \\
&= \sum_{n=0}^{\infty}\sum_{m=0}^{\infty} c_n^* c_m E_m \int \psi_n^*(\boldsymbol{x})\psi_m(\boldsymbol{x})\,d\boldsymbol{x} \\
&= \sum_{n=0}^{\infty} E_n |c_n|^2 \\
&\geq E_0 \sum_{n=0}^{\infty} |c_n|^2 = E_0 \int \psi^*(\boldsymbol{x})\psi(\boldsymbol{x})\,d\boldsymbol{x}
\end{aligned} \tag{4.56}$$

so therefore $E[\psi] \ge E_0$, with the minimum occurring when $\psi = \psi_0$. This suggests that the ground state can be found by minimizing the energy functional over a family of wave functions. However, the wave function $\psi(\boldsymbol{x})$ is a continuous function of position, and when we express it as a set of discrete values on an N-point spatial grid the minimization problem becomes a problem of minimization on an N-dimensional space. This can be solved using Monte Carlo methods.

The technique presented here is quite simple. An initial guess of the wave function is made on a discrete grid of points. One of these points is selected at random and adjusted by a random amount to give us a trial wave function. The energy function of Eq. (4.53) is evaluated for this trial wave function and is compared to the energy function evaluated for the original guess: if the energy is larger for the trial wave function then original guess is retained; otherwise the guess is updated with the trial wave function. Note that we wish to have a normalized wave function so it is helpful to normalize the waveform after making adjustments.

We illustrate the method with the application of finding the (known) ground state of the quantum harmonic oscillator given a (rather poor) initial guess. The harmonic oscillator potential is

$$V(x) = \frac{1}{2} kx^2 \tag{4.57}$$

where k is the spring constant and the Hamiltonian operator is

$$\hat{H} = -\frac{\hbar^2}{2m}\frac{\partial^2}{\partial x^2} + V(x). \tag{4.58}$$

For simplicity we adopt units in which $k = 1$, $m = 1$, and $\hbar = 1$. The one-dimensional domain is gridded into N evenly spaced points centered at the origin with spacing $\Delta x = (x_{\text{max}} - x_{\text{min}})/(N-1)$ where $x_{\text{max}} = 5$ and $x_{\text{min}} = -5$ in our units. Our initial guess for ψ_i is constant throughout the domain except at the endpoints, ψ_0 and ψ_{N-1} where we set it to zero $\psi_0 = \psi_{N-1} = 0$; the endpoints are kept fixed at zero throughout the procedure. The constant value is chosen so that the wave function is normalized,

$$\Delta x \sum_{i=1}^{N-2} |\psi_i|^2 = 1. \tag{4.59}$$

To produce an updated trial wave function, one of the points $j = 1, 2, \dots, N-2$ is selected at random and the value of the wave function at that point is multiplied by a factor that is drawn from a uniform distribution between 0.8 and 1.2 and the resulting trial waveform is then normalized. Equation (4.53) is then computed for the trial wave function,

$$E[\psi^{\text{trial}}] = \Delta x \sum_{i=1}^{N-2} \psi_i^{\text{trial}} \hat{H} \psi_i^{\text{trial}} \tag{4.60}$$

where

$$\hat{H}\psi_i^{\text{trial}} = -\frac{\hbar^2}{2m}\frac{\psi_{i+1}^{\text{trial}} - 2\psi_i^{\text{trial}} + \psi_{i-1}^{\text{trial}}}{(\Delta x)^2} + V_i \psi_i^{\text{trial}}. \tag{4.61}$$

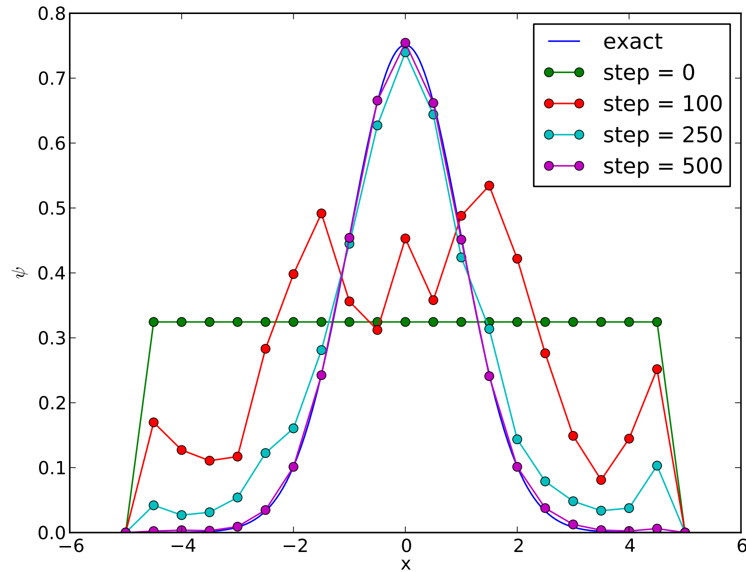

Figure 4.8: Snapshots of some of the intermediate states explored during a run of the program `variational.py` for $N = 21$ points in the interval $-5 \leq x \leq 5$.

The program `variational.py` in Listing 4.4 implements these methods to compute the ground state wave function and energy for the quantum harmonic oscillator. As seen in Fig. 4.8, a reasonably accurate approximation to the true ground state wave function is achieved after a few hundred updates and Fig. 4.9 shows that the correct ground state energy is rapidly found.

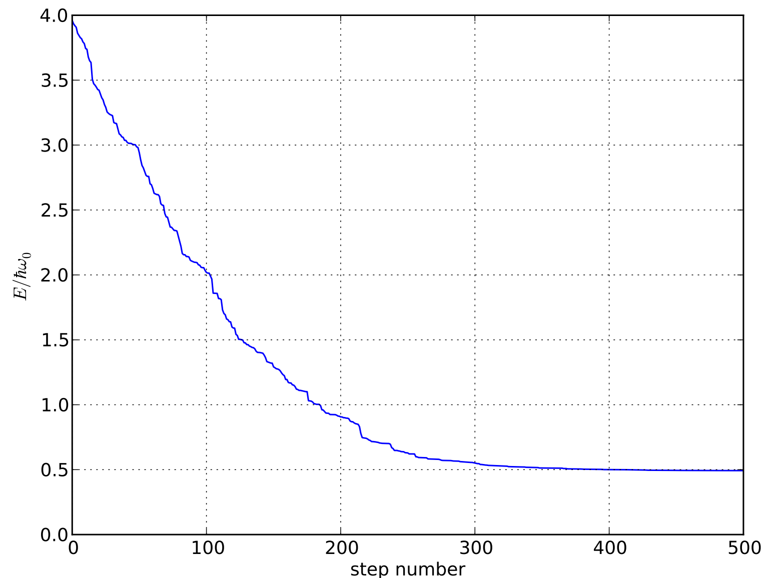

Figure 4.9: The value of the energy functional as a function of update step number for a run of the program `variational.py` for $N = 21$ points in the interval $-5 \leq x \leq 5$. The true ground state $\frac{1}{2}\hbar\omega_0$ is found with good accuracy.

Listing 4.4: Program `variational.py`

```
1  import math, random, pylab
2  
3  # parameters for harmonic oscillator
4  hbar = 1.0
5  m = 1.0
6  k = 1.0
7  omega0 = (k/m)**0.5
8  
9  # input number of grid points, steps, random seed
10 N = input('number of grid points -> ')
11 nstep = input('number of steps -> ')
12 seed = input('random number seed -> ')
13 random.seed(seed)
14 
15 # setup grid and initial guess
16 xmin = -5.0
17 xmax = 5.0
18 dx = (xmax-xmin)/(N-1)
19 x = pylab.arange(xmin, xmax+0.1*dx, dx)
20 psi = pylab.ones(N)   # initial guess
```

```
21 psi[0] = psi[N−1] = 0.0   # endpoints fixed at zero
22 
23 # compute energy, potential, normalization
24 V = pylab.zeros(N)
25 E = pylab.zeros(nstep+1)
26 ssq = 0
27 for i in range(1, N−1):
28     V[i] = k*x[i]**2/2.0
29     H = −hbar**2*(psi[i−1]−2.0*psi[i]+psi[i+1])/(2*m*dx**2)
30     H += V[i]*psi[i]
31     E[0] += psi[i]*H*dx
32     ssq += psi[i]**2*dx
33 E[0] /= ssq
34 psi /= ssq**0.5
35 
36 # prepare animated plot
37 pylab.ion()
38 xfine = pylab.arange(xmin, xmax, 0.01)
39 psi0 = [(m*omega0/(math.pi*hbar))**0.25*math.exp(−0.5*m*omega0*xx**2/
       hbar)
40         for xx in xfine]
41 pylab.plot(xfine, psi0)
42 (line, ) = pylab.plot(x, psi, 'o−')
43 pylab.ylabel('$\psi$')
44 pylab.xlabel('x')
45 
46 # perform the evolution
47 n = 1
48 while n <= nstep:
49     # choose a random point and a random amount to change psi
50     tmp = pylab.copy(psi)   # temporary wavefunction trial
51     j = random.choice(range(1, N−1))
52     tmp[j] *= random.uniform(0.8, 1.2)
53 
54     # normalize and compute energy
55     E[n] = 0.0
56     ssq = 0.0
57     for i in range(1, N−1):
58         H = −hbar**2*(tmp[i−1]−2.0*tmp[i]+tmp[i+1])/(2*m*dx**2)
59         H += V[i]*tmp[i]
60         E[n] += tmp[i]*H*dx
61         ssq += tmp[i]**2*dx
62     E[n] /= ssq
63 
64     # test if the trial wavefunction reduces energy
65     if E[n] < E[n−1]:
66         # update current wavefunction
67         psi = tmp/ssq**0.5
68 
69         # update plot
```

```
70          line.set_ydata(psi)
71          pylab.title('%d moves'%n)
72          pylab.draw()
73
74          # increment step count
75          n += 1
76
77  # freeze animation and plot energy as a function of time
78  pylab.ioff()
79  pylab.figure()
80  pylab.plot(range(nstep+1), E)
81  pylab.ylabel('$E / \hbar\omega_0$')
82  pylab.xlabel('step number')
83  pylab.grid()
84  pylab.show()
```

Exercise 4.3 Use the variational method to find a numerical approximation to the ground state wave function and energy for (a) a V-shaped potential $V(x) = |x|$, and (b) the quartic potential $V(x) = x^4$.

Chapter 5

Data reduction

5.1 Statistical description of data

The *measures of central tendency* of a set of data are such statistics as the *mean*, the *median*, and the *mode*, as well as various moments of the data. The sample mean of a set of N points, $\{x_i\}$ where $i = 0, 1, 2, \ldots, N-1$, is simply

$$\bar{x} = \frac{1}{N}\sum_{i=0}^{N-1} x_i. \tag{5.1}$$

A straightforward implementation is as follows:

```
>>> data = [3, 2, 5, 5, 2, 0, 3, 5, 2, 1, 0, 7]
>>> mean = sum(data) / float(len(data))
>>> mean
2.9166666666666665
```

The mean of the data makes use of the value of each of the data points, which means that if one of these values is extreme — either very large or very small — then the mean can be skewed by this single point. The median and mode are more robust because they make fewer requirements on the values of the data points.

The median simply requires that the values of the data points have some ordination, that is, we only need to be able to determine if x_i is greater than, less than, or equal to x_j, but not by how much. The median is then the value that is in the middle. (If the number of data points is even, there are two values in the middle; in such a situation it is common to take the mean of these two values.) Given a set of data $\{x_i\}$, reorder the data to form the *order statistics* $\{x_{(0)}, x_{(1)}, \ldots, x_{(N-1)}\}$ where $x_{(0)}$ is the smallest value, $x_{(1)}$ is the second smallest value, etc., and $x_{(N-1)}$ is the largest value. The median is then

$$\tilde{x} = \begin{cases} x_{((N-1)/2)} & \text{if } N \text{ is odd} \\ \frac{1}{2}(x_{(N/2-1)} + x_{(N/2)}) & \text{if } N \text{ is even.} \end{cases} \tag{5.2}$$

In addition, the *statistical range* is $R = x_{(N-1)} - x_{(0)}$ and the *midrange* is $MR = \frac{1}{2}(x_{(N-1)} + x_{(0)})$. We can find the median by

```
>>> data = [3, 2, 5, 5, 2, 0, 3, 5, 2, 1, 0, 7]
>>> data.sort()
>>> median = 0.5 * (data[(len(data) - 1)//2] + data[len(data)//2])
>>> median
2.5
```

Finally, the mode requires only that values of the data points can be put into categories. The frequency of each category is the number of data points whose values are in that category, and the mode is the category with the highest frequency. It is possible that more than one category share this highest frequency in which case the data is *multimodal*. The mode can be obtained as follows

```
>>> data = [3, 2, 5, 5, 2, 0, 3, 5, 2, 1, 0, 7]
>>> vals = set(data)
>>> freq = [data.count(v) for v in vals]
>>> maxfreq = max(freq)
>>> mode = [v for i, v in enumerate(vals) if freq[i] == maxfreq]
>>> mode
[2, 5]
```

Note that in this example, the data is bimodal with modes 2 and 5.

To illustrate that the median makes fewer assumptions about the nature of the data than the mean, and the mode makes even fewer than the median, consider the following: The median requires only an ordinal relation (except if the number of data points is even), so we can find the median of such things as, say, letters:

```
>>> data = list('abracadabra')
>>> data.sort()
>>> median = data[len(data)//2]
>>> median
'b'
```

Note that we know that the letter b comes after the letter a, and c after b, but we do not need to assign a numerical value to each letter in order to get the median. Meanwhile, the mode does not even require an ordinal relationship between the data points. For example, we do not know whether an apple is bigger or smaller than an orange or a banana, but we can still find the mode of a basket of fruit:

```
>>> data = ['orange', 'apple', 'banana', 'apple', 'apple', 'orange']
>>> vals = set(data)
>>> freq = [data.count(v) for v in vals]
>>> maxfreq = max(freq)
>>> mode = [v for i, v in enumerate(vals) if freq[i] == maxfreq]
>>> mode
['apple']
```

The mean, median, and mode all are measures of the central value of the data. We often want to have a measure of the spread of the data as well. The *standard deviation* is

$$s = \sqrt{\frac{1}{N-1}\sum_{i=0}^{N-1}(x_i - \bar{x})^2}. \tag{5.3}$$

Notice the factor $1/(N-1)$ is used when the mean $\bar{x}$ is computed from the same data as is used to compute the standard deviation; if the value of the mean is known *a priori* then the factor $1/N$ should be used. The standard deviation could be computed as follows

```
>>> data = [3, 2, 5, 5, 2, 0, 3, 5, 2, 1, 0, 7]
>>> mean = sum(data) / float(len(data))
>>> sdev = (sum((v-mean)**2 for v in data)/(len(data)-1.0))**0.5
>>> mean, sdev
(2.9166666666666665, 2.1933093855190746)
```

The standard deviation is typically used in conjunction with the mean as measures of the central value and the spread. When the median is used as the measure of the central value, a measure of the spread that is in the same spirit would be the *interquartile range*, which is the range of values that span the second and third quartiles. In terms of the order statistics, the first quartile is $Q_1 = x_{(\lfloor N/4 \rfloor)}$, the third quartile is $Q_3 = x_{(\lfloor (3N-1)/4 \rfloor)}$, and the interquartile range is $Q_3 - Q_1$. Here, $\lfloor a \rfloor$ is the greatest integer less than or equal to a. For example,

```
>>> data = list('abracadabra')
>>> data.sort()
>>> median = data[len(data)//2]
>>> q1 = data[len(data)//4]
>>> q3 = data[(3*len(data)-1)//4]
>>> q1, median, q3
('a', 'b', 'd')
```

This shows that the median letter is b, and that at least half of the letters are in the range between a and d inclusive. Finally, when the mode is used, the spread is often characterized in terms of the *full width at half maximum* or FWHM, which is all the categories that have a frequency at least half of the maximum frequency:

```
>>> data = ['orange', 'apple', 'banana', 'apple', 'apple', 'orange']
>>> vals = set(data)
>>> freq = [data.count(v) for v in vals]
>>> maxfreq = max(freq)
>>> mode = [v for i, v in enumerate(vals) if freq[i] == maxfreq]
>>> fwhm = [v for i, v in enumerate(vals) if freq[i] > maxfreq // 2]
>>> mode, fwhm
(['apple'], ['orange', 'apple'])
```

That is, apples are the most common fruit, but the number of oranges is not less than half of the number of apples.

We can also define these statistics for continuous distributions. A *probability density function*, $f(x)$, is a function that is everywhere non-negative, $f(x) \geq 0$ for all x, and is normalized,

$$\int_{-\infty}^{\infty} f(x)\,dx = 1. \tag{5.4}$$

We say that the probability of x having a value between x and $x + dx$ for small dx is $f(x)\,dx$. The mean of the distribution is the expected value of x, $\langle x \rangle$,

$$\mu = \langle x \rangle = \int_{-\infty}^{\infty} x\,f(x)\,dx, \tag{5.5}$$

and the standard deviation of the distribution is

$$\sigma = \sqrt{\mathrm{Var}(x)} \tag{5.6}$$

where the *variance* of the distribution is

$$\mathrm{Var}(x) = \langle (x-\mu)^2 \rangle = \langle x^2 \rangle - \langle x \rangle^2 \tag{5.7}$$

with

$$\langle x^2 \rangle = \int_{-\infty}^{\infty} x^2 f(x)\,dx. \tag{5.8}$$

The *cumulative distribution* is

$$F(x) = \int_{-\infty}^{x} f(x')\,dx' \tag{5.9}$$

and the probability of X being below a value x is $\Pr\{X \leq x\} = F(x)$. The inverse function of the cumulative distribution function is the *quantile function*, $Q(p)$, which satisfies $Q(p) = x$ for the value of x that satisfies $F(x) = p$:

$$Q(p) = \inf\{x \mid p \leq F(x)\}. \tag{5.10}$$

In terms of the quantile function, the median is $\mu_{1/2} = Q(1/2)$, the first and third quartiles are $Q_1 = Q(1/4)$ and $Q_3 = Q(3/4)$, and the interquartile range is $\mathrm{IQR} = Q_3 - Q_1$.

The mode of the distribution is the point (or points) of its maximum value,

$$\mathrm{mode} = \arg\max_x f(x) \tag{5.11}$$

where

$$\arg\max_x f(x) = \{x \mid \forall y : f(y) \leq f(x)\}. \tag{5.12}$$

The half-maximum points are

$$H_1 = \inf\{x \mid f(x) \geq \tfrac{1}{2} f_{\max}\} \tag{5.13}$$

$$H_2 = \sup\{x \mid f(x) \geq \tfrac{1}{2} f_{\max}\} \tag{5.14}$$

where $f_{\max} = \max_x f(x)$, and the full width at half maximum is $\text{FWHM} = H_2 - H_1$.

For example, consider the Cauchy distribution,

$$f(x; x_0, \gamma) = \frac{1}{\pi} \frac{\gamma}{(x - x_0)^2 + \gamma^2}, \tag{5.15}$$

which has a maximum value of $f_{\max} = 1/(\pi\gamma)$ at $x_{\text{mode}} = x_0$ and is half-maximum at the values $x_0 \pm \gamma$, so the FWHM $= 2\gamma$. The cumulative distribution function for the Cauchy distribution is

$$F(x; x_0, \gamma) = \frac{1}{2} + \frac{1}{\pi} \arctan\left(\frac{x - x_0}{\gamma}\right) \tag{5.16}$$

and the quantile function is therefore

$$Q(p; x_0, \gamma) = x_0 + \gamma \tan[\pi(p - \tfrac{1}{2})] \tag{5.17}$$

from which we find that the median is $\mu_{1/2} = x_0$ and the first and third quartiles are $Q_1 = x_0 - \gamma$ and $Q_3 = x_0 + \gamma$. The inter-quartile range is therefore IQR $= 2\gamma$. Hence we see that for the Cauchy distribution, the median and the mode have the same value, x_0, and the full width at half maximum and the inter-quartile range also have the same value, 2γ. Interestingly, the mean and the variance cannot be computed.

Another example is the *normal distribution* or *Gaussian distribution*,

$$f(x; \mu, \sigma) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x - \mu)^2}{2\sigma^2}\right). \tag{5.18}$$

As the parameter symbols suggest, the mean of the distribution is μ and its standard deviation is σ. The mode is also equal to the mean, and the full-width at half-maximum is $2\sqrt{2\ln 2}\sigma \approx 2.355\sigma$. The cumulative distribution function is

$$F(x; \mu, \sigma) = \frac{1}{2}\left[1 + \operatorname{erf}\left(\frac{x - \mu}{\sqrt{2\sigma^2}}\right)\right] \tag{5.19}$$

where $\operatorname{erf}(x)$ is the *error function*,

$$\operatorname{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt. \tag{5.20}$$

The median is equal to the mean, $\mu_{1/2} = \mu$, and the quartiles are $Q_1 = \mu + \sigma z_1$ and $Q_2 = \mu + \sigma z_3$ where $z_1 \approx -0.67449$ and $z_3 \approx +0.67449$ are the *standard scores* of the first and third quartiles, so the interquartile range is IQR $\approx 1.349\sigma$.

Often we wish to compute the mean and the standard deviation of a number of samples having to keep all values in the dataset. Perhaps the samples are being continually produced and we want to have a current estimate of the mean and

standard deviation at every time (i.e., a running mean and a running standard deviation), or perhaps we simply do not want to have to store all the values. We can achieve this as follows: first note that

$$n \overset{n}{\bar{x}} = \sum_{i=0}^{n-1} x_i \tag{5.21}$$

where $\overset{n}{\bar{x}}$ is the average of the first n points. Then,

$$(n+1) \overset{n+1}{\bar{x}} = n \overset{n}{\bar{x}} + x_n \tag{5.22}$$

so we have

$$\overset{n+1}{\bar{x}} = \overset{n}{\bar{x}} + \frac{x_n - \overset{n}{\bar{x}}}{n+1}. \tag{5.23}$$

Using this equation, we can continually update the running mean as more samples are accumulated. Now consider the sum of the squares of the deviations from the *current* mean when there are n samples,

$$\overset{n}{M}_2 = \sum_{i=0}^{n-1} \left(x_i - \overset{n}{\bar{x}} \right)^2. \tag{5.24}$$

When we add an additional sample, we want to compute

$$\begin{aligned}
\overset{n+1}{M}_2 &= \sum_{i=0}^{n} \left(x_i - \overset{n+1}{\bar{x}} \right)^2 \\
&= \sum_{i=0}^{n-1} \left(x_i - \overset{n+1}{\bar{x}} \right)^2 + \left(x_n - \overset{n+1}{\bar{x}} \right)^2 \\
&= \sum_{i=0}^{n-1} \left[\left(x_i - \overset{n}{\bar{x}} \right) - \left(\overset{n+1}{\bar{x}} - \overset{n}{\bar{x}} \right) \right]^2 + \left(x_n - \overset{n+1}{\bar{x}} \right) \left[\left(x_n - \overset{n}{\bar{x}} \right) - \left(\overset{n+1}{\bar{x}} - \overset{n}{\bar{x}} \right) \right] \\
&= \sum_{i=0}^{n-1} \left\{ \left(x_i - \overset{n}{\bar{x}} \right)^2 - 2\left(x_i - \overset{n}{\bar{x}} \right)\left(\overset{n+1}{\bar{x}} - \overset{n}{\bar{x}} \right) + \left(\overset{n+1}{\bar{x}} - \overset{n}{\bar{x}} \right)^2 \right\} \\
&\quad + \left(x_n - \overset{n+1}{\bar{x}} \right)\left(x_n - \overset{n}{\bar{x}} \right) - \left(x_n - \overset{n+1}{\bar{x}} \right)\left(\overset{n+1}{\bar{x}} - \overset{n}{\bar{x}} \right) \\
&= \sum_{i=0}^{n-1} \left(x_i - \overset{n}{\bar{x}} \right)^2 + n\left(\overset{n+1}{\bar{x}} - \overset{n}{\bar{x}} \right)^2 \\
&\quad + \left(x_n - \overset{n+1}{\bar{x}} \right)\left(x_n - \overset{n}{\bar{x}} \right) - \left(x_n - \overset{n+1}{\bar{x}} \right)\left(\overset{n+1}{\bar{x}} - \overset{n}{\bar{x}} \right) \\
&= \overset{n}{M}_2 + \left(x_n - \overset{n+1}{\bar{x}} \right)\left(x_n - \overset{n}{\bar{x}} \right) + \left[n\left(\overset{n+1}{\bar{x}} - \overset{n}{\bar{x}} \right) - \left(x_n - \overset{n+1}{\bar{x}} \right) \right]\left(\overset{n+1}{\bar{x}} - \overset{n}{\bar{x}} \right) \\
&= \overset{n}{M}_2 + \left(x_n - \overset{n+1}{\bar{x}} \right)\left(x_n - \overset{n}{\bar{x}} \right) + \left[(n+1) \overset{n+1}{\bar{x}} - n \overset{n}{\bar{x}} - x_n \right]\left(\overset{n+1}{\bar{x}} - \overset{n}{\bar{x}} \right)
\end{aligned} \tag{5.25}$$

The term in the square brackets on the last line vanishes according to Eq. (5.22), and so we have the formula

$$\overset{n+1}{M}_2 = \overset{n}{M}_2 + \Big(x_n - \overset{n+1}{\bar{x}}\Big)\Big(x_n - \overset{n}{\bar{x}}\Big) \tag{5.26}$$

for updating the running value of M_2. The standard deviation when there are n samples is then

$$\overset{n}{s} = \left(\frac{\overset{n}{M}_2}{n-1}\right)^{1/2}. \tag{5.27}$$

Now suppose that we have *two* sets of data, $\{x_i\}$ and $\{y_i\}$, with $i = 0, 1, 2, \ldots, N-1$. If we want to determine if the two sets of data are correlated, we can compute the *linear correlation coefficient* or *Pearson's* r,

$$r = \frac{\sum_{i=0}^{N-1}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=0}^{N-1}(x_i - \bar{x})^2}\sqrt{\sum_{i=0}^{N-1}(y_i - \bar{y})^2}}. \tag{5.28}$$

The range of values of r are between -1 and $+1$. If the value of r is close to zero then we say the data sets are *uncorrelated*; if it is close to $+1$ we say the data sets are *correlated*; if it is close to -1 we say the data sets are *anti-correlated*. A recurrence method for computing the numerator of the correlation coefficient for n samples of (x, y) pairs,

$$\overset{n}{C} = \sum_{i=0}^{n-1}\Big(x_i - \overset{n}{\bar{x}}\Big)\Big(y_i - \overset{n}{\bar{y}}\Big) \tag{5.29}$$

is

$$\overset{n+1}{C} = \overset{n}{C} + \Big(x_n - \overset{n}{\bar{x}}\Big)\Big(y_n - \overset{n+1}{\bar{y}}\Big) \tag{5.30}$$

or, equivalently,

$$\overset{n+1}{C} = \overset{n}{C} + \Big(x_n - \overset{n+1}{\bar{x}}\Big)\Big(y_n - \overset{n}{\bar{y}}\Big). \tag{5.31}$$

Then the value of $\overset{n}{r}$ for the nth pair is

$$\overset{n}{r} = \frac{\overset{n}{C}}{\sqrt{\overset{n}{M}_{2,x}\overset{n}{M}_{2,y}}}. \tag{5.32}$$

Now we want to know whether a computed value of r is *significant*. Let us assume a *null hypothesis*, that the observed data samples $\{x_i\}$ and $\{y_i\}$ are random numbers taken from independent distributions. In particular, we assume that these independent probability distributions are normal distributions. The means and standard deviations of the underlying Gaussian distribution from which the x values were drawn are in general different from the mean and standard deviation of the Gaussian distribution from which the y values were drawn, but because the distributions are independent we would not expect any correlation between the x

values and the y values under our null hypothesis. Given the null hypothesis we can then compute the probability density function for the value of Pearson's r given N pairs of samples drawn from these independent Gaussian distributions. It is

$$f(r;N) = \frac{1}{B\left(\frac{1}{2}, \frac{N-2}{2}\right)} (1-r^2)^{(N-4)/2} \tag{5.33}$$

where

$$B(a,b) = \int_0^1 t^{a-1}(1-t)^{b-1} dt = \frac{\Gamma(a)\Gamma(b)}{\Gamma(a+b)} \tag{5.34}$$

is known as the *Euler beta function*. It is related to the *gamma function*, which, for integer arguments, is given by $\Gamma(n) = (n-1)!$, and for half-integer arguments it is

$$\Gamma(n+\tfrac{1}{2}) = \frac{1 \cdot 3 \cdot 5 \cdots (2n-1)}{2^n}\sqrt{\pi} = \frac{(n-1)!!}{2^n}\sqrt{\pi}. \tag{5.35}$$

Using this distribution we can compute the probability p of getting a value of r with $|r| > r_p$ for some number r_p:

$$p = 1 - \int_{-r_p}^{r_p} f(r;N)\,dr. \tag{5.36}$$

To evaluate this integral, we perform a change of variables to $x = r^2$; we then find

$$p = 1 - \frac{1}{B\left(\frac{1}{2}, \frac{N-2}{2}\right)} \int_0^{r_p^2} x^{-1/2}(1-x)^{(N-4)/2}\,dx = 1 - \frac{B\left(r_p^2; \frac{1}{2}, \frac{N-2}{2}\right)}{B\left(\frac{1}{2}, \frac{N-2}{2}\right)} \tag{5.37}$$

where

$$B(x;a,b) = \int_0^x t^{a-1}(1-t)^{b-1} dt \tag{5.38}$$

is known as the *incomplete beta function*. Therefore, when we calculate Pearson's r for our data sets, we can measure its *p-value*, which is the probability of obtaining a value at least as extreme if the null hypothesis is true. A very small p-value, which would occur for a value of r close to $+1$ or -1, would lead us to reject the null hypothesis and conclude that correlation is statistically significant.

As an example, consider the period-luminosity relationship for Cepheid variables, which are a category of variable star that undergoes regular oscillations in its brightness. The period of these oscillations is correlated with the overall luminosity of the star. This data is given in Table 5.1 and is stored in a file `cepheids.dat` (Listing 5.1). The program `cepheid.py` then reads this file and computes the correlation coefficient between the logarithm of the period and the magnitude and also produces a plot of the data, which is shown in Fig. 5.1. The correlation coefficient is found to be $r = -0.894$, which for $N = 10$ data points has a p-value of $p = 0.05\%$ so the null hypothesis (of uncorrelated data sets) is quite unlikely. (Note that brighter stars have *smaller* magnitudes, so it is conventional to plot the vertical magnitude axis increasing *downwards*.) Rather than performing a numerical integral to compute the incomplete beta function, the function `betainc` in the `scipy.special` module is used.

Period (days)		Magnitude	
P	ΔP	V	ΔV
40.9345	1.0315	22.985	0.008
75.3739	2.2386	22.676	0.016
49.4143	2.3786	23.290	0.019
42.6833	3.6532	23.412	0.044
30.8813	0.7128	23.532	0.014
27.2601	0.4105	23.626	0.016
21.7926	0.3649	23.669	0.021
30.4055	0.7575	23.699	0.015
30.4193	0.5237	23.728	0.016
22.8373	0.1005	23.776	0.013

Table 5.1: Period, P, and V-band magnitude, V data, with uncertainties ΔP and ΔV for 10 Cepheid variables in the galaxy M101. This data is from Benjamin J. Shappee and K. Z. Stanek "A new Cepheid distance to the giant spiral M101 based on image subtraction of *Hubble Space Telescope*/Advanced Camera for Surveys observations," The Astrophysical Journal, **733** 124 (2011).

Listing 5.1: Data file `cepheids.dat`

```
# P     dP      V       dV
40.9345 1.0315  22.985  0.008
75.3739 2.2386  22.676  0.016
49.4143 2.3786  23.290  0.019
42.6833 3.6532  23.412  0.044
30.8813 0.7128  23.532  0.014
27.2601 0.4105  23.626  0.016
21.7926 0.3649  23.669  0.021
30.4055 0.7575  23.699  0.015
30.4193 0.5237  23.728  0.016
22.8373 0.1005  23.776  0.013
```

Listing 5.2: Program `cepheid.py`

```python
import pylab, scipy.special, math

# read file cepheids.dat and extract data
(P, dP, V, dV) = pylab.loadtxt('cepheids.dat', unpack=True)
N = len(P)  # number of data points

# compute linear correlation coefficient
x = [math.log10(P[i]) for i in range(N)]
y = V
meanx = 0.0
meany = 0.0
```

```
M2x = 0.0
M2y = 0.0
C = 0.0
for i in range(N):
    deltax = x[i]-meanx
    deltay = y[i]-meany
    meanx = meanx+deltax/(i+1)
    meany = meany+deltay/(i+1)
    M2x = M2x+deltax*(x[i]-meanx)
    M2y = M2y+deltay*(y[i]-meany)
    C = C+deltax*(y[i]-meany)
r = C/(M2x*M2y)**0.5
p = 1-scipy.special.betainc(0.5, 0.5*(N-2), r**2)
print 'correlation coefficient, r = %f'%r
print 'probability under null hypothesis, p = %f%%'%(100*p)

# plot the data
pylab.errorbar(P, V, fmt='o', xerr=dP, yerr=dV)
pylab.ylim(reversed(pylab.ylim()))  # reverse y-axis
pylab.xscale('log')
pylab.xlabel('Period (days)')
pylab.ylabel('Magnitude')
pylab.title('Period-magnitude relation for Cepheid variables in M101')
pylab.grid(which='both')
pylab.show()
```

5.2 Statistical tests of data distribution

Often, given a set of data, we wish to know if it is consistent with either a hypothesized distribution or with another set of data. For example, we might try to determine if a die is fair by rolling it some large number of times and see if the probability of getting each result is the same. As another example, suppose we wish to determine if a set of random data has a Gaussian distribution (i.e., we wish to know if it was drawn from a Gaussian distribution). In the first example the data can have only discrete values and we commonly use a *Pearson's chi-squared test*. In the second case the data values are continuous and we the *Kolmogorov-Smirnov test* or *K-S test* could be used. (Note that data whose values are continuous can be converted into data whose values are discrete by binning the data.)

We will first consider the K-S test to see if data is drawn from a particular continuous distribution with a known cumulative distribution function $F(x)$. From the N data points, $\{x_i\}$ for $i = 0, 1, 2, \ldots, N-1$, construct the *empirical distribution function*

$$F_N(x) = \frac{1}{N}\sum_{i=0}^{N-1}\Theta(x - x_i) \tag{5.39}$$

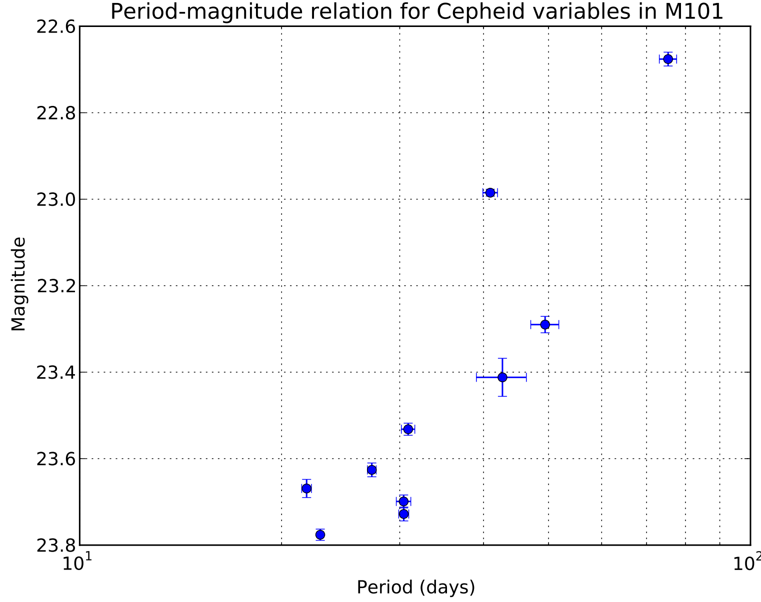

Figure 5.1: Results from running the program `cepheid.py` (Listing 5.2) which inputs the data file `cepheids.dat` (Listing 5.1, see also Table 5.1) and computes the correlation coefficent r and its p-value p. The values $r = -0.894$ and $p = 0.05\%$ are found showing that period and magnitude are strongly anticorrelated.

where $\Theta(x)$ is the *Heaviside step function*,

$$\Theta(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0. \end{cases} \tag{5.40}$$

The K-S test statistic is the largest distance between the empirical distribution function and the hypothesized distribution function,

$$D = \max_x |F_N(x) - F(x)|. \tag{5.41}$$

A large value of D indicates that the empirical distribution function is dissimilar to the hypothesized distribution function and would lead us to reject the null hypothesis (that the data points were drawn from the hypothesized distribution). In the limit of large N, under the null hypothesis, the distribution of the quantity $K = \sqrt{N}D$ is known and so a p-value can be computed from the formula

$$p = 2\sum_{i=1}^{\infty}(-1)^{i-1}\mathrm{e}^{-2i^2K^2}. \tag{5.42}$$

For moderate values of N this formula for the p-value is approximately correct with

$$K = \left(\sqrt{N} + 0.12 + \frac{0.11}{\sqrt{N}}\right) D. \tag{5.43}$$

As an example, suppose that we have N data points that are drawn from a normal distribution with mean μ and standard deviation σ, while our null hypothesis is that they are drawn from a normal distribution with zero mean and unit standard deviation. The K-S test can be used to test the null hypothesis. Figure 5.2 shows the results from the program `ks1.py` which simulates this scenario. For the particular choice of parameters, with $N = 13$, $\mu = 0$, and $\sigma = 3$ (and random seed of 101) we find $D = 0.424$ which is very significant having a p-value of only $p = 1.26\%$.

Listing 5.3: Program `ks1.py`

```
1  import pylab, random, math
2  
3  N = input('number of data points -> ')
4  mu = input('mean of true distribution -> ')
5  sigma = input('standard deviation of true distribution -> ')
6  seed = input('random number seed -> ')
7  random.seed(seed)
8  
9  # generate data
10 x = [random.gauss(mu, sigma) for i in range(N)]
11 
12 # compute K-S test statistic
13 x.sort()
14 D = 0.0  # biggest difference
15 for i in range(N):
16     # the hypothesized cumulative distribution at this point
17     F = 0.5*(1.0+math.erf(x[i]/2**0.5))
18     F0 = i/float(N)
19     F1 = (i+1)/float(N)
20 
21     # compute distance both before and after this point
22     d = max(abs(F-i/float(N)), abs(F-(i+1)/float(N)))
23 
24     # keep this value if it is the largest so far
25     if d > D:
26         (D, X) = (d, x[i])
27         if abs(F-F0) > abs(F-F1):  # determine the interval
28             segment = [F, F0]
29         else:
30             segment = [F, F1]
31 
32 # compute p-value for this statistic
33 K = D*(N**0.5+0.12+0.11/N**0.5)
34 a = 2.0
35 b = -2.0*K**2
```

```
36 p = 0.0
37 eps = 1e-6
38 for i in range(1, 20):
39     term = a*math.exp(b*i**2)
40     if abs(term) < eps*p:
41         break
42     p += term
43     a *= -1.0
44
45 # plot the empirical and hypothetical distributions
46 xfine = pylab.arange(-3.0, 3.0, 0.01)
47 F = [0.5*(1.0+math.erf(xx/2**0.5)) for xx in xfine]
48 pylab.plot(xfine, F, label='hypothetical')
49 pylab.plot(x+[float('inf')], [i/float(N) for i in range(N+1)], 'o-',
50            drawstyle='steps', label='empirical')
51 pylab.plot([X, X], segment, 'D--', label='D = %g'%D)
52 pylab.ylabel('cumulative probability')
53 pylab.xlabel('x')
54 pylab.title('p-value = %3g%%'%(100*p))
55 pylab.legend(loc=0)
56 pylab.grid()
57 pylab.show()
```

The K-S test can also be used to test if two sets of data, $\{x_{1,i}\}$ for $i = 0,1,2,\ldots,N_1-1$ and $\{x_{2,j}\}$ for $j = 0,1,2,\ldots,N_2-1$, were drawn from the same distribution (the null hypothesis). Now there are two empirical distribution functions, $F_{N_1}(x)$ and $F_{N_2}(x)$, and the test statistic is

$$D = \max_x |F_{N_1}(x) - F_{N_2}(x)|. \tag{5.44}$$

Furthermore, the approximate p-value can be obtained as before using the quantity K in Eq. (5.43) but now N is an effective number of data points, which is the *harmonic sum* of the number of data points in each set,

$$\frac{1}{N} = \frac{1}{N_1} + \frac{1}{N_2} \tag{5.45}$$

or

$$N = \frac{N_1 N_2}{N_1 + N_2}. \tag{5.46}$$

An example of the use of the K-S test applied to two data sets is given in the program `ks2.py`. Here the two sets of data are both drawn from Cauchy distributions but with different medians x_0 and scale parameters γ. Figure 5.3 shows the results when the medians of the distributions are both 0 but the scale parameters are different with $\gamma_1 = 0.1$ and $\gamma_2 = 0.5$. The first data set has $N_1 = 13$ samples and the second data set has $N_2 = 10$ samples. The random seed is 101. The K-S test statistic is found to be $D = 0.52$ and the p-value is only $p = 5.8\%$. Notice that to generate the random data samples drawn from the Cauchy distribution we use the quantile function of Eq. (5.17) with the value p being a uniform deviate in the range from 0 to 1.

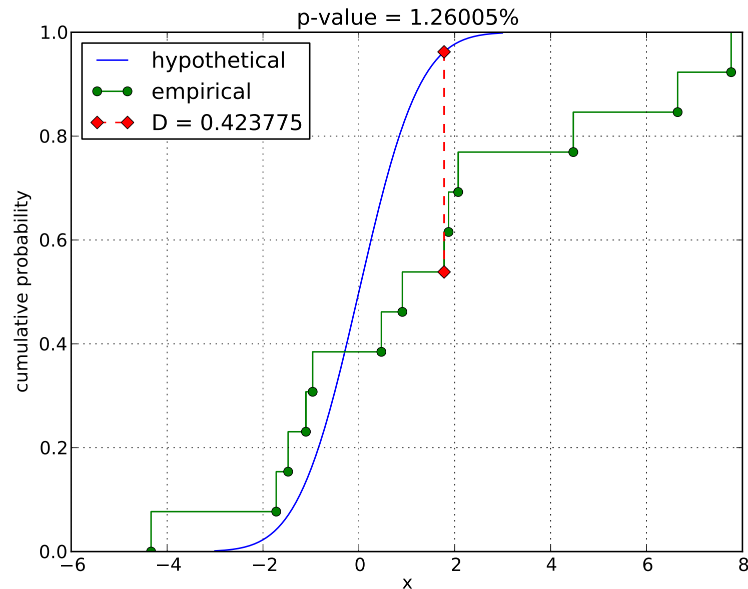

Figure 5.2: Results from running the program `ks1.py` (Listing 5.3) with the input parameters $N = 13$, $\mu = 0$, $\sigma = 3$, and random number seed 101. The hypothetical (zero-mean and unit-variance null hypothesis) and empirical cumulative distributions are shown and the red line segment shows the maximum interval between these two distributions (which gives D). The p-value is very low, $p = 1.26\%$, which shows there little support for the null hypothesis.

Listing 5.4: Program `ks2.py`

```
1  import pylab, random, math
2  
3  N1 = input('number of data points in set 1 -> ')
4  N2 = input('number of data points in set 2 -> ')
5  med1 = input('median of set 1 distribution -> ')
6  med2 = input('median of set 2 distribution -> ')
7  gam1 = input('scale parameter of set 1 distribution -> ')
8  gam2 = input('scale parameter of set 2 distribution -> ')
9  seed = input('random number seed -> ')
10 random.seed(seed)
11 
12 # generate data
13 x1 = [med1+gam1*math.tan(math.pi*(random.random()-0.5)) for i in range(
       N1)]
14 x2 = [med2+gam2*math.tan(math.pi*(random.random()-0.5)) for i in range(
       N2)]
15 
```

```
16 # compute K-S test statistic
17 x1.sort()
18 x2.sort()
19 x1 += [float('inf')]  # add a final point at infinity
20 x2 += [float('inf')]  # add a final point at infinity
21 F1 = F2 = 0.0
22 D = 0.0  # biggest difference
23 i = j = 0
24 while i <= N1 and j <= N2:
25     F1 = i/float(N1)
26     F2 = j/float(N2)
27     d = abs(F1-F2)
28     if x1[i] < x2[j]:
29         x = x1[i]
30         i += 1
31     elif x1[i] > x2[j]:
32         x = x2[j]
33         j += 1
34     else:
35         x = x1[i]
36         i += 1
37         j += 1
38     if d > D:
39         (D, X) = (d, x)
40         segment = [F1, F2]
41
42 # compute p-value for this statistic
43 N = N1*N2/float(N1+N2)  # effective number of data points
44 K = D*(N**0.5+0.12+0.11/N**0.5)
45 a = 2.0
46 b = -2.0*K**2
47 p = 0.0
48 eps = 1e-6
49 for i in range(1, 20):
50     term = a*math.exp(b*i**2)
51     if abs(term) < eps*p:
52         break
53     p += term
54     a *= -1.0
55
56 # plot the two distributions
57 pylab.plot(x1, [i/float(N1) for i in range(N1+1)], 'o-', drawstyle='
       steps',
58            label='x1')
59 pylab.plot(x2, [i/float(N2) for i in range(N2+1)], 'o-', drawstyle='
       steps',
60            label='x2')
61 pylab.plot([X, X], segment, 'D--', label='D = %g'%D)
62 pylab.ylabel('cumulative probability')
63 pylab.xlabel('x')
```

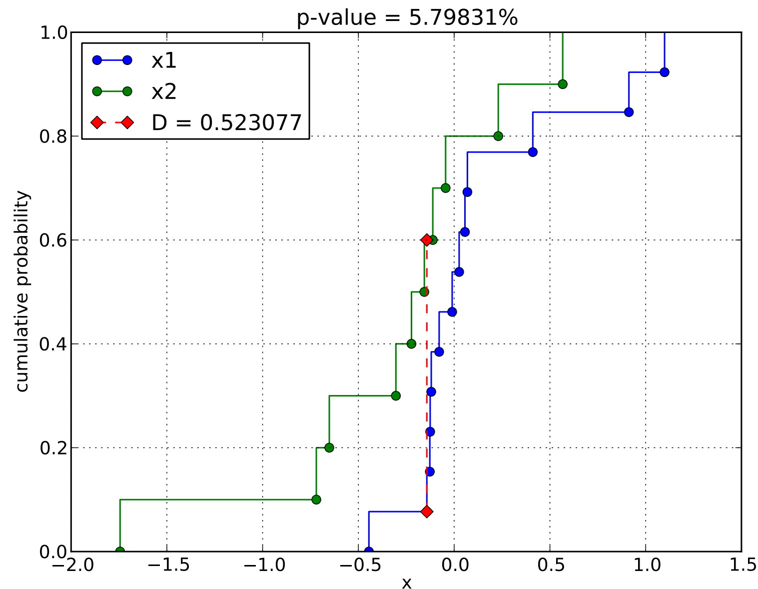

Figure 5.3: Results from running the program `ks2.py` (Listing 5.4) with the input parameters $N_1 = 13$, $N_2 = 10$, $x_{0,1} = 0$, $x_{0,2} = 0$, $\gamma_1 = 0.1$, $\gamma_2 = 0.5$, and random number seed 101. The two empirical distributions are shown and the red line segment shows the maximum interval between these two distributions (which gives D). The p-value is small, $p = 5.6\%$.

```
64 pylab.title('p−value = %3g%%'%(100*p))
65 pylab.legend(loc=2)
66 pylab.grid()
67 pylab.show()
```

Now we consider Pearson's chi-squared statistic for categorical data. As a model problem, consider an experiment attempting to determine if a hypothetical event rate (e.g., radioactive decay events) is consistent with observation event counts. In a given measurement time τ the expected number of events from a Poisson process is $\mu = \lambda\tau$ where λ is the event rate. Given this Poisson mean, μ, the probability of obtaining n events is

$$f(n;\mu) = \frac{\mu^n}{n!}\,e^{-\mu}. \tag{5.47}$$

Suppose we perform N measurements of a Poisson process with mean μ and we obtain a set of event counts, $\{n_i\}$ for $i = 0, 1, 2, \ldots N-1$. The observed frequencies are $\{O_k\}$ for $k = 0, 1, 2, \max\{n_i\}$ where O_k is the number of times that $n_i = k$ in the

N measurements,

$$O_k = \sum_{i=0}^{N-1} \delta_{n_i,k}. \tag{5.48}$$

The expected frequencies are

$$E_k = N f(k;\mu_0) = N \frac{\mu_0^k}{k!}\, e^{-\mu_0} \qquad k = 0,1,2,\ldots,\max\{n_i\}. \tag{5.49}$$

where μ_0 is the hypothesized Poisson mean under the null hypothesis. Pearson's chi-squared statistic is then given by

$$X^2 = \sum_{k=0}^{K-1} \frac{(O_k - E_k)^2}{E_k} \tag{5.50}$$

where K is the number of bins.

We need to choose the number of bins to use in constructing the observed and expected frequencies. A normal choice of the number of bins to use is $K = \max\{n_i\} + 2$, where the bins $k = 0,1,2,\ldots,\max\{n_i\}$ all correspond frequencies of obtaining k events, and the last bin $k = \max\{n_i\} + 1$ corresponds to the frequency of obtaining more than $\max\{n_i\}$ events. We have obviously $O_{K-1} = 0$ and

$$E_{K-1} = N - \sum_{k=0}^{K-2} E_k. \tag{5.51}$$

Now we need to determine the probability of obtaining a value X^2 or greater under the null hypothesis. If we consider the sum

$$\chi^2 = \sum_{i=0}^{\nu-1} z_i^2 \tag{5.52}$$

where each of the z_i values are drawn from a normal distribution with zero-mean and unit-variance then χ^2 follows a *chi-squared distribution*,

$$f(\chi^2;\nu) = \frac{1}{2^{\nu/2}\Gamma(\nu/2)}\,(\chi^2)^{\nu/2-1} e^{-\chi^2/2}. \tag{5.53}$$

The p-value for a given χ^2 is then

$$p = 1 - \frac{\gamma(\nu/2, \chi^2/2)}{\Gamma(\nu/2)} \tag{5.54}$$

where

$$\gamma(a,x) = \int_0^x t^{a-1} e^{-t}\, dt \tag{5.55}$$

is the *incomplete gamma function*. If we approximate each term in the sum of Eq. (5.50) for X^2 as being approximately the square of a normal deviate of zero

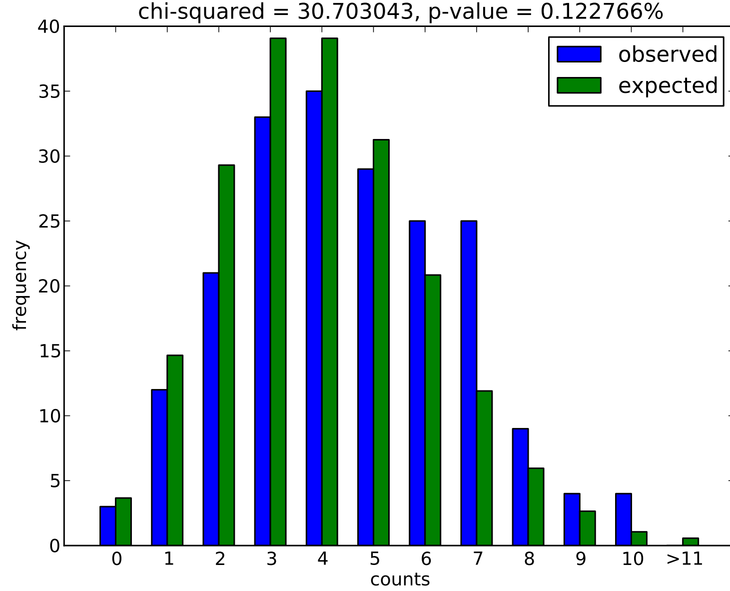

Figure 5.4: Results from running the program `chisq1.py` (Listing 5.5) with the input parameters $N = 200$ measurments, $\mu = 4.5$ (the true Poisson mean), $\mu_0 = 4$ (the hypothesized Poisson mean), and random number seed 101. The observed and expected frequencies are shown. The test statistic is $X^2 = 30.7$ and its p-value for $\nu = 11$ degrees of freedom ($K = 12$ bins) is $p = 0.12\%$.

mean and unit variance then we would expect X^2 to be chi-squared distributed under the null hypothesis. However, not every term in the sum is independent since we have the constraint that the sum of the values of E_k must equal the sum of the values of O_k. This constraint means that the number of degrees of freedom that we must use is $\nu = K - 1$ in determining the significance of a measured X^2 value.

The program `chisq1.py` performs a simulation in which N measurements are made of a Poisson process of true mean μ, and then the chi-squared test statistic and its p-value are computed for a null hypothesis of a Poisson process of mean μ_0. To simulate this scenario, we need to construct a method for choosing the observed number of events, n_i, for each measurement i. For a Poisson process of rate λ, probability distribution that the next event will occur in time t is $f(t;\lambda) = \lambda \exp(-\lambda\tau)$. We can then generate events at times

$$t_j - t_{j-1} = -\lambda^{-1} \log u_j \tag{5.56}$$

where $t_0 = 0$ and u_j is a uniform deviate between 0 and 1, and continue while $t_j < \tau$. A trick to doing this efficiently is rather than adding exponentially-distributed deviates until the sum is greater than τ, multiply uniform random devi-

ates until the product is less than $\exp(-\mu)$ where $\mu = \lambda\tau$ is the Poisson mean. The program `chisq1.py` is given in Listing 5.5, and Fig. 5.4 shows the observed frequencies $N = 200$ observations of a process with a true mean $\mu = 4.5$ and the expected frequencies for the hypothetical mean $\mu_0 = 4$. The test statistic is $X^2 = 30.7$ and its p-value for $\nu = 11$ degrees of freedom ($K = 12$ bins) is $p = 0.12\%$ which indicates that the observed frequencies are not consistent with the expected frequencies. The random number seed 101 is used.

Listing 5.5: Program `chisq1.py`

```
import pylab, scipy.special, random, math

N = input('number of measurements -> ')
mu = input('true rate of events -> ')
mu0 = input('rate of events under null hypothesis -> ')
seed = input('random number seed -> ')
random.seed(seed)

# generate data
n = [0]*N   # number of events for each measurement
q = math.exp(-mu)
for i in range(N):
    p = random.random()
    while p > q:
        p *= random.random()
        n[i] += 1

# compute observed and expected distributions
K = max(n)+2   # number of bins; last one is the >max(n) bin
E = [0.0]*K   # expected frequency
O = [0]*K   # observed frequency
factorial = 1   # k!
for k in range(K-1):
    O[k] = n.count(k)
    E[k] = N*mu0**k*math.exp(-mu0)/factorial
    factorial *= k+1
# remaining number in the >max(n) bin
E[K-1] = N-sum(E)

# compute chi-squared statistic
chisq = sum((O[k]-E[k])**2/E[k] for k in range(K))

# compute significance
nu = K-1   # degrees of freedom
p = 1.0-scipy.special.gammainc(0.5*nu, 0.5*chisq)

# plot results
counts = pylab.array(range(K))
pylab.bar(counts-0.3, O, color='b', width=0.3, label='observed')
pylab.bar(counts, E, color='g', width=0.3, label='expected')
```

```
41 labels = [str(k) for k in range(K)]
42 labels[K-1] = '>'+labels[K-1]
43 pylab.xticks(counts, labels)
44 pylab.ylabel('frequency')
45 pylab.xlabel('counts')
46 pylab.xlim(xmin=-1, xmax=K)
47 pylab.legend()
48 pylab.title('chi-squared = %f, p-value = %f%%'%(chisq, 100.0*p))
49 pylab.show()
```

Pearson's chi-squared test can also be used to determine if two data sets are drawn from the same distribution. Let $\{R_k\}$, $k = 0, 1, 2, \ldots, K-1$ be the observed frequencies from the first data set and $\{S_k\}$, $k = 0, 1, 2, \ldots, K-1$ be the observed frequencies from the second data set. Pearson's chi-squared test statistic is now

$$X^2 = \sum_{k=0}^{K-1} \frac{\left(\sqrt{N_2/N_1}\,R_k - \sqrt{N_1/N_2}\,S_k\right)^2}{R_k + S_k}. \tag{5.57}$$

where

$$N_1 = \sum_{k=0}^{K-1} R_k \quad \text{and} \quad N_2 = \sum_{k=0}^{K-1} S_k. \tag{5.58}$$

Here we only include those bins k in which $R_k + S_k \neq 0$.

The program `chisq2.py` performs a simulation in which the first data set comprises N_1 measurements of a Poisson process with mean μ_1 from source 1 and the second data set comprises N_2 measurements of a Poisson process with mean μ_2 from source 2. The result when $N_1 = 200$, $N_2 = 300$, $\mu_1 = 5$, $\mu_2 = 4$, with random seed 101 is a test statistic value $X^2 = 41$ for $\nu = 11$ degrees of freedom ($K = 12$ bins) which has a very small p-value, $p = 0.002\%$.

Listing 5.6: Program `chisq2.py`

```
1  import pylab, scipy.special, random, math
2
3  N1 = input('number of measurements from source 1 -> ')
4  N2 = input('number of measurements from source 2 -> ')
5  mu1 = input('rate of events from source 1 -> ')
6  mu2 = input('rate of events from source 2 -> ')
7  seed = input('random number seed -> ')
8  random.seed(seed)
9
10 # generate data
11 n1 = [0]*N1  # number of events for each measurement from source 1
12 n2 = [0]*N2  # number of events for each measurement from source 2
13 q1 = math.exp(-mu1)
14 q2 = math.exp(-mu2)
15 for i in range(N1):
16     p = random.random()
17     while p > q1:
```

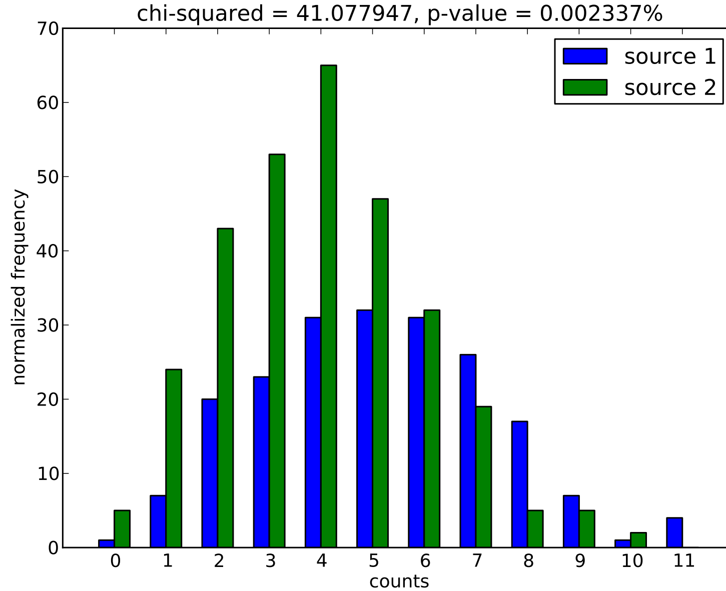

Figure 5.5: Results from running the program `chisq2.py` (Listing 5.6) with the input parameters $N_1 = 200$ measurments from a Poisson process with mean $\mu_1 = 5$ as the first set of data and $N_2 = 300$ measurements from a Poisson process with mean $\mu_2 = 4$, and a random number seed 101. The value of the test statistic is $X^2 = 41$ for $\nu = 11$ degrees of freedom ($K = 12$ bins) and a p-value of $p = 0.002\%$.

```
18            p *= random.random()
19            n1[i] += 1
20    for i in range(N2):
21        p = random.random()
22        while p > q2:
23            p *= random.random()
24            n2[i] += 1
25
26    # compute observed and expected distributions
27    K = max(n1+n2)+1
28    R = [n1.count(k) for k in range(K)]   # frequencies from source 1
29    S = [n2.count(k) for k in range(K)]   # frequencies from source 1
30    counts = range(K)
31
32    # delete bins where R and S are both zero; note: need to go backwards
33    for k in reversed(range(K)):
34        if R[k] == 0 and S[k] == 0:
35            del R[k]
```

```
36            del S[k]
37            del counts[k]
38
39    # remaining number of bins
40    K = len(counts)
41
42    # compute chi-squared statistic
43    Q1 = (float(N2)/float(N1))**0.5
44    Q2 = 1.0/Q1
45    chisq = sum((Q1*R[k]-Q2*S[k])**2/(R[k]+S[k]) for k in range(K))
46
47    # compute significance
48    nu = K-1   # degrees of freedom
49    p = 1.0-scipy.special.gammainc(0.5*nu, 0.5*chisq)
50
51    # plot results
52    counts = pylab.array(counts)
53    pylab.bar(counts-0.3, R, color='b', width=0.3, label='source 1')
54    pylab.bar(counts, S, color='g', width=0.3, label='source 2')
55    pylab.ylabel('normalized frequency')
56    pylab.xlabel('counts')
57    pylab.xticks(range(min(counts), max(counts)+1))
58    pylab.xlim(xmin=min(counts)-1, xmax=max(counts)+1)
59    pylab.legend()
60    pylab.title('chi-squared = %f, p-value = %f%%'%(chisq, 100.0*p))
61    pylab.show()
```

5.3 Data modelling

Suppose that we have a set of N data points $\{(x_i, y_i)\}$, $i = 0, 1, 2, \ldots, N-1$ which we wish to fit to some function $y = f(x; \boldsymbol{a})$ where $\boldsymbol{a}$ are the parameter of the function that we are trying to determine. Suppose that the uncertainties in the values y_i are σ_i and for now we will suppose that there are no uncertainties in the values of x_i (or that these uncertainties are very small). If our model is correct then we suppose that the residuals,

$$e_i = y_i - f(x_i; \boldsymbol{a}), \tag{5.59}$$

are normally-distributed random numbers with zero mean and variance σ_i. In this case, the quantity

$$\chi^2 = \sum_{i=0}^{N-1} \frac{e_i^2}{\sigma_i^2} = \sum_{i=0}^{N-1} \frac{[y_i - f(x_i; \boldsymbol{a})]^2}{\sigma_i^2} \tag{5.60}$$

will be chi-squared distributed with $\nu = N$ degrees of freedom. To determine the best fit parameters $\hat{\boldsymbol{a}}$ we minimize the value χ^2 over the parameters. That is, we

find the parameter values that solve the M equations,

$$0 = \sum_{i=0}^{N-1} \frac{y_i - f(x_i; \hat{\boldsymbol{a}})}{\sigma_i^2} \frac{\partial f(x_i; \hat{\boldsymbol{a}})}{\partial a_j} \qquad j = 0, 1, 2, \ldots, M-1 \tag{5.61}$$

for the M parameter values $\{a_j\}$, $j = 0, 1, 2, \ldots, M-1$. This is known as *chi-square fitting*.

As a concrete example, consider fitting data to a straight line, which is known as *linear regression*. The equation of a straight line is

$$y = f(x; a, b) = a + bx \tag{5.62}$$

where a and b are the intercept and the slope respectively. The equations we must solve are then

$$0 = \sum_{i=0}^{N-1} \frac{y_i - \hat{a} - \hat{b}x_i}{\sigma_i} \tag{5.63}$$

and

$$0 = \sum_{i=0}^{N-1} \frac{x_i(y_i - \hat{a} - \hat{b}x_i)}{\sigma_i}. \tag{5.64}$$

These equations can be written as the system of equations

$$\begin{aligned} \hat{a}S + \hat{b}S_x &= S_y \\ \hat{a}S_x + \hat{b}S_{xx} &= S_{xy} \end{aligned} \tag{5.65}$$

where

$$\begin{aligned} S &= \sum_{i=0}^{N-1} \frac{1}{\sigma_i^2} & S_x &= \sum_{i=0}^{N-1} \frac{x_i}{\sigma_i^2} & S_y &= \sum_{i=0}^{N-1} \frac{y_i}{\sigma_i^2} \\ S_{xx} &= \sum_{i=0}^{N-1} \frac{x_i^2}{\sigma_i^2} & S_{xy} &= \sum_{i=0}^{N-1} \frac{x_i y_i}{\sigma_i^2} \end{aligned} \tag{5.66}$$

and we obtain the solution

$$\hat{a} = \frac{S_{xx}S_y - S_x S_{xy}}{\Delta} \qquad \hat{b} = \frac{S S_{xy} - S_x S_y}{\Delta} \tag{5.67}$$

with

$$\Delta = S S_{xx} - S_x^2. \tag{5.68}$$

We can further estimate the errors in the fit parameters $\hat{a}$ and $\hat{b}$. By linear error propagation, these will be

$$\sigma_a^2 = \sum_{i=0}^{N-1} \left(\frac{\partial \hat{a}}{\partial y_i} \sigma_i \right)^2 \quad \text{and} \quad \sigma_b^2 = \sum_{i=0}^{N-1} \left(\frac{\partial \hat{b}}{\partial y_i} \sigma_i \right)^2. \tag{5.69}$$

We evaluate the partial derivatives using Eq. (5.67),

$$\frac{\partial \hat{a}}{\partial y_i} = \frac{1}{\sigma_i^2}\frac{S_{xx} - x_i S_x}{\Delta} \qquad \frac{\partial \hat{b}}{\partial y_i} = \frac{1}{\sigma_i^2}\frac{x_i S - S_x}{\Delta}, \tag{5.70}$$

and we obtain

$$\sigma_a^2 = \frac{S_{xx}}{\Delta} \qquad \sigma_b^2 = \frac{S}{\Delta}. \tag{5.71}$$

In addition, we can compute the covariance between $\hat{a}$ and $\hat{b}$, σ_{ab}, which is

$$\sigma_{ab} = \sum_{i=0}^{N-1} \frac{\partial f(x_i;\hat{a},\hat{b})}{\partial \hat{a}} \frac{\partial f(x_i;\hat{a},\hat{b})}{\partial \hat{b}} \sigma_i^2 = -\frac{S_x}{\Delta}. \tag{5.72}$$

The value of χ^2 for the best-fit parameters, $\hat{a}$ and $\hat{b}$, then determines the *goodness of fit* of our model, which is the p-value for that value of χ^2,

$$p = 1 - \frac{\gamma(\nu/2, \chi^2/2)}{\Gamma(\nu/2)} \tag{5.73}$$

where in this case the number of degrees of freedom is $\nu = N - 2$ since there are two fitted parameters. If p is very small then the fit is questionable (or the errors are underestimated).

The program `regress.py` demonstrates the above procedure for linear regression. The data points are generated with $y_i = a + bx_i + e_i$ where e_i are random normally-distributed errors in the independent variable with variance σ_i^2 where the values of σ_i are themselves randomly chosen with a log-normal distribution, $f(\sigma) = (2\pi\sigma^2)^{1/2}\exp(-\frac{1}{2}\ln^2\sigma)$ then scaled to 10% of their value. Figure 5.6 shows the results of the program `regress.py` for $N = 10$ data points with true intercept and slope parameters $a = 0$ and $b = 1$ respectively and a random number seed of 101; the estimated intercept and slope are $\hat{a} = -0.01 \pm 0.05$ and $\hat{b} = 1.02 \pm 0.09$ respectively and the fit has $\chi^2 = 6.35$ and a p-value of $p = 61\%$, which indicates that the fit is good.

Listing 5.7: Program `regress.py`

```
1  import pylab, scipy.special, random
2  
3  N = input('number of points -> ')
4  a = input('true intercept -> ')
5  b = input('true slope -> ')
6  seed = input('random seed -> ')
7  random.seed(seed)
8  
9  # generate random data with random uncertainties in y
10 x = pylab.arange(0.5/N, 1.0, 1.0/N)
11 dy = [0.1*random.lognormvariate(0.0, 1.0) for i in range(N)]
12 y = [random.gauss(a+b*x[i], dy[i]) for i in range(N)]
13 
```

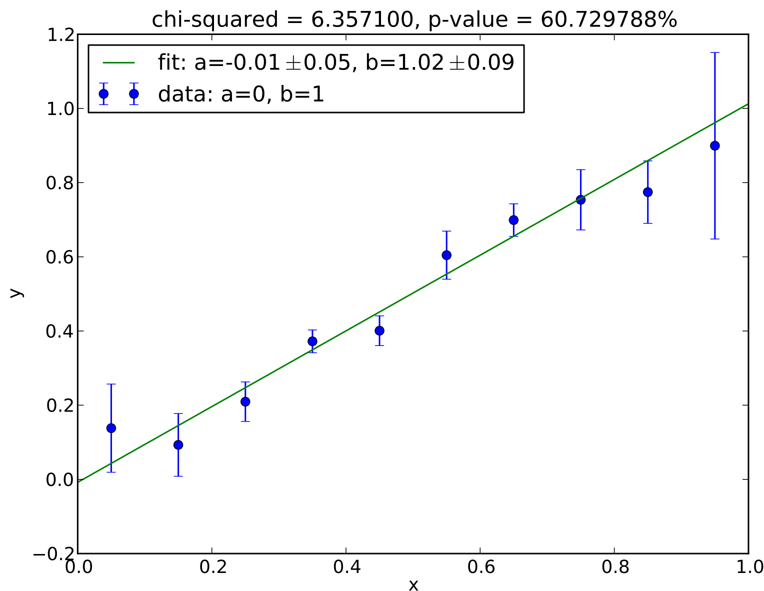

Figure 5.6: Results from running the program `regress.py` (Listing 5.7) with input parameters $N = 10$, $a = 0$, $b = 1$, and random number seed 101. The resulting fit has $\hat{a} = -0.01 \pm 0.05$, $\hat{b} = 1.02 \pm 0.09$, $\chi^2 = 6.35$, and p-value $p = 61\%$.

```
14 # compute linear regression coefficients ahat and bhat and uncertainties
15 w = [1.0/dy[i]**2 for i in range(N)]
16 S = sum(w[i] for i in range(N))
17 Sx = sum(w[i]*x[i] for i in range(N))
18 Sy = sum(w[i]*y[i] for i in range(N))
19 Sxx = sum(w[i]*x[i]**2 for i in range(N))
20 Sxy = sum(w[i]*x[i]*y[i] for i in range(N))
21 Delta = S*Sxx-Sx**2
22 ahat = (Sxx*Sy-Sx*Sxy)/Delta
23 bhat = (S*Sxy-Sx*Sy)/Delta
24 da = (Sxx/Delta)**0.5
25 db = (S/Delta)**0.5
26
27 # compute chi-square and p-value of the fit
28 chisq = sum(w[i]*(y[i]-ahat-bhat*x[i])**2 for i in range(N))
29 p = 1.0-scipy.special.gammainc((N-2.0)/2.0, chisq/2.0)
30
31 # plot the data and the fit
32 pylab.errorbar(x, y, fmt='o', yerr=dy, label='data: a=%g, b=%g'%(a, b))
33 pylab.plot([0.0, 1.0], [ahat, ahat+bhat],
```

```
34            label='fit: a=%.2f$\pm$%.2f, b=%.2f$\pm$%.2f'%(ahat, da, bhat
        , db))
35 pylab.xlabel('x')
36 pylab.ylabel('y')
37 pylab.title('chi-squared = %f, p-value = %f%%'%(chisq, 100.0*p))
38 pylab.legend(loc=2)
39 pylab.show()
```

Exercise 5.1 The period and magnitude data for Cepheid variables given in Table 5.1 can be used to determine the distance to M101. From observations of nearby Cepheid variables, it is known that the period-luminosity relationship is given by

$$M_V = -1.43 - 2.81 \log_{10}\left(\frac{P}{1\ \text{day}}\right) \tag{5.74}$$

where M_V is known as the *absolute magnitude*, which is the magnitude that a star would have if it were located at a distance of ten *parsecs* (10 pc) where

$$1\,\text{pc} = 3.085\,680\,25 \times 10^{16}\,\text{m}. \tag{5.75}$$

The relationship between apparent magnitude, V, absolute magnitude, M_V, and distance d is

$$d = 10^{(V-M_V+5)/5}\,\text{pc} \tag{5.76}$$

where $V - M_V$ is known as the *distance modulus*. By fitting a straight line to the Cepheid data of V versus $\log_{10}(P/1\,\text{day})$, one can determine the distance to M101.

The data in Table 5.1 has errors in both the measured values of the magnitude as well as the measured values of the period. When both the independent and dependent variables contain uncertainties $\{\sigma^2_{x,i}\}$ and $\{\sigma^2_{y,i}\}$ respectively, the variance in the residuals $e_i = y_i - a - bx_i$ of a linear fit will be

$$\sigma_i^2 = \text{Var}(y_i - a - bx_i) = \sigma^2_{y,i} + b^2\sigma^2_{x,i} \tag{5.77}$$

and a chi-squared fit requires finding the values of a and b for which

$$\chi^2 = \sum_{i=0}^{N-1} \frac{(y_i - a - bx_i)^2}{\sigma^2_{y,i} + b^2\sigma^2_{x,i}} \tag{5.78}$$

is a minimum. The presence of the slope parameter b in the denominator makes the problem of linear regression considerably more difficult.

However, the period-luminosity relationship for Cepheid variables has a known slope, $b = -2.81$. Therefore, one needs only to find the intercept a in the equation

$$V = a - 2.81 \log_{10}\left(\frac{P}{1\ \text{day}}\right) \tag{5.79}$$

for the Cepheid data in order to determine the distance to M101.

Use Eq. (5.78) to obtain a formula for the minimum chi-squared estimate of the intercept a, and, using the Cepheid data, find the intercept in Eq. (5.79). Then, using the period-luminosity relationship for Cepheids, Eq. (5.74), and the distance modulus formula of Eq. (5.76), determine the distance in parsecs to M101. (The uncertainty in $x = \log_{10}(P/1\,\text{day})$, σ_x, is related to the uncertainty in P, σ_P, by $\sigma_x = \sigma_P/(P \ln 10)$.)

We now go beyond fitting data to a line and consider the problem of fitting data to a more complicated function. We suppose that the function that we wish to fit to the data is a linear combination of M basis functions, $f_i(x)$, $j = 0, 1, 2, \ldots, M-1$, so that

$$y = \sum_{j=0}^{M-1} a_j f_j(x) \tag{5.80}$$

where the values of the coefficients a_j are to be determined. For example we may wish to fit the data to a polynomial

$$y = a_0 + a_1 x + a_2 x^2 + \cdots + a_{M-1} x^{M-1}. \tag{5.81}$$

The *general linear least squares* fit then minimizes

$$\chi^2 = \sum_{i=0}^{N-1} \frac{\left(y_i - \sum_{j=0}^{M-1} a_j f_j(x_i)\right)^2}{\sigma_i^2} \tag{5.82}$$

for the parameters a_j. This equation can be written in matrix form as

$$\chi^2 = \|\mathbf{X} \cdot \boldsymbol{a} - \boldsymbol{b}\|^2 \tag{5.83}$$

where $\boldsymbol{a}$ is a M-dimensional column vector whose components are the parameters a_j, $\boldsymbol{b}$ is an N-dimensional column vector whose components are the independent variables weighted by the inverse of their standard deviations, $b_i = y_i/\sigma_i$, and $\mathbf{X}$ is a $N \times M$ *design matrix* of the weighted basis functions $X_{ij} = f_j(x_i)/\sigma_i$:

$$\chi^2 = \left\| \begin{bmatrix} \frac{f_0(x_0)}{\sigma_0} & \frac{f_1(x_0)}{\sigma_0} & \cdots & \frac{f_{M-1}(x_0)}{\sigma_0} \\ \frac{f_0(x_1)}{\sigma_1} & \frac{f_1(x_1)}{\sigma_1} & \cdots & \frac{f_{M-1}(x_1)}{\sigma_1} \\ \frac{f_0(x_2)}{\sigma_2} & \frac{f_1(x_2)}{\sigma_2} & \cdots & \frac{f_{M-1}(x_2)}{\sigma_2} \\ \vdots & \vdots & & \vdots \\ \frac{f_0(x_{N-1})}{\sigma_{N-1}} & \frac{f_1(x_{N-1})}{\sigma_{N-1}} & \cdots & \frac{f_{M-1}(x_{N-1})}{\sigma_{N-1}} \end{bmatrix} \cdot \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{M-1} \end{bmatrix} - \begin{bmatrix} \frac{y_0}{\sigma_0} \\ \frac{y_1}{\sigma_1} \\ \frac{y_2}{\sigma_2} \\ \vdots \\ \frac{y_{N-1}}{\sigma_{N-1}} \end{bmatrix} \right\|^2 \tag{5.84}$$

Notice that if the number of parameters M is equal to the number of data points N then the design matrix $\mathbf{X}$ is a square matrix so we can simply solve the linear system $\mathbf{X} \cdot \boldsymbol{a} = \boldsymbol{b}$ for $\boldsymbol{a}$ by the methods described in Sec. A.1. The resulting fit is then "perfect" with $\chi^2 = 0$. This is not the interesting case, however: our goal is to find the best fit model having fewer parameters than data points. In this case the matrix $\mathbf{X}$ is not a square matrix and we cannot invert it and solve for $\boldsymbol{a}$.

Our goal is to minimize χ^2 and over the unknown parameters $\boldsymbol{a}$ and thereby determine the parameters of the best fit $\hat{\boldsymbol{a}}$. At the minimum point we have $\nabla_{\boldsymbol{a}}\chi^2 = 0$, which results in the *normal equations*

$$\mathbf{X}^\mathsf{T} \cdot \mathbf{X} \cdot \hat{\boldsymbol{a}} = \mathbf{X}^\mathsf{T} \cdot \boldsymbol{b} \tag{5.85}$$

where $\mathbf{X}^\mathsf{T}$ is the *transpose* of the matrix $\mathbf{X}$. Notice that the matrix $\mathbf{X}^\mathsf{T} \cdot \mathbf{X}$ is a $M \times M$ square matrix, $\boldsymbol{a}$ is a $M \times 1$ column vector, and $\mathbf{X}^\mathsf{T} \cdot \boldsymbol{b}$ is also a $M \times 1$ column vector. Therefore we can solve the normal equations for $\hat{\boldsymbol{a}}$ using the methods described in Sec. A.1. The solution is

$$\hat{\boldsymbol{a}} = \boldsymbol{\Sigma} \cdot \mathbf{X}^\mathsf{T} \cdot \boldsymbol{b} \tag{5.86}$$

where $\boldsymbol{\Sigma} = (\mathbf{X}^\mathsf{T} \cdot \mathbf{X})^{-1}$ is the covariance matrix for the parameters. That is, the variance in $\hat{a}_j$, σ_{a_j}, is the component Σ_{jj}^2 of the matrix $\boldsymbol{\Sigma}$. This can be seen by computing

$$\sigma_{a_j}^2 = \sum_{i=0}^{N-1} \left(\frac{\partial \hat{a}_j}{\partial y_i} \sigma_i \right)^2 \tag{5.87}$$

where

$$\frac{\partial \hat{a}_j}{\partial y_i} \sigma_i = \frac{\partial \hat{a}_j}{\partial b_i} = \sum_{k=0}^{M-1} \Sigma_{jk} X_{ik}, \tag{5.88}$$

which results in

$$\sigma_{a_j}^2 = \sum_{i=0}^{N-1} \sum_{k=0}^{M-1} \sum_{l=0}^{M-1} \Sigma_{jk} \Sigma_{jl} X_{ik} X_{il} = \sum_{k=0}^{M-1} \sum_{l=0}^{M-1} \Sigma_{jk} \Sigma_{jl} (\Sigma^{-1})_{kl} = \sum_{k=0}^{M-1} \Sigma_{jk} \delta_{jk} = \Sigma_{jj}. \tag{5.89}$$

The covariance matrix $\boldsymbol{\Sigma}$ specifies a confidence ellipsoid in parameter space. If it is not diagonal, then the uncertainties in the parameters are correlated and principal axes of the ellipsoid will be linear combinations of the parameters. To interpret what a confidence ellipsoid means, note that the parameters that minimize χ^2, $\hat{\boldsymbol{a}}$, will not be exactly the true value of the parameters, but will be within some distance $\Delta\boldsymbol{a}$ of the true parameters. The difference between the value of χ^2 for the *true* parameters and the minimum value of χ^2 will be

$$\begin{aligned} \Delta\chi^2 &= \|\mathbf{X} \cdot (\hat{\boldsymbol{a}} + \Delta\boldsymbol{a}) - \boldsymbol{b}\|^2 - \|\mathbf{X} \cdot \hat{\boldsymbol{a}} - \boldsymbol{b}\|^2 \\ &\approx \Delta\boldsymbol{a}^\mathsf{T} \cdot \mathbf{X}^\mathsf{T} \cdot \mathbf{X} \cdot \Delta\boldsymbol{a} \\ &= \Delta\boldsymbol{a}^\mathsf{T} \cdot \boldsymbol{\Sigma}^{-1} \cdot \Delta\boldsymbol{a} \end{aligned} \tag{5.90}$$

where we have used the fact that χ^2 is a minimum for the parameters $\hat{\boldsymbol{a}}$. It turns out that $\Delta\chi^2$ is chi-squared distributed with M degrees of freedom if the errors in the original data set are normally distributed. In this case, we can construct an ellipsoid of constant $\Delta\chi^2$ that will contain the true value of the parameters with probability $1 - p$ where

$$1 - p = \frac{\gamma(M/2, \Delta\chi^2/2)}{\Gamma(M/2)} \tag{5.91}$$

$1-p$ (%)	$M=1$	$M=2$	$M=3$
68.27	1.00	2.30	3.53
90	2.71	4.61	6.25
95	3.84	5.99	7.82
95.45	4.00	6.18	8.03
99	6.64	9.21	11.34
99.73	9.00	11.83	14.16

Table 5.2: $\Delta\chi^2$ values for confidence ellipsiods of confidence level $1-p$ for the joint estimation of M parameters in the large data sample limit. Data from Table 33.2 of the 2011 Review of Particle Physics, K. Nakamura et al. (Particle Data Group), Journal of Physics G **37** 075021 (2010).

is known as the *confidence level* for the ellipsoid. Values of $\Delta\chi^2$ corresponding to various confidence levels are given in Table 5.2.

Once a confidence level is chosen and its $\Delta\chi^2$ value is found, the error ellipsoid would be the locus of points $\boldsymbol{a}$ in parameter space that satisfies

$$(\boldsymbol{a}-\hat{\boldsymbol{a}})^{\mathsf{T}} \cdot \boldsymbol{\Sigma}^{-1} \cdot (\boldsymbol{a}-\hat{\boldsymbol{a}}) = \Delta\chi^2. \tag{5.92}$$

In order to determine this ellipsoid, we perform a *Cholesky decomposition* of the matrix $\boldsymbol{\Sigma}$,

$$\boldsymbol{\Sigma} = \mathbf{L} \cdot \mathbf{L}^{\mathsf{T}} \tag{5.93}$$

where $\mathbf{L}$ is a lower-diagonal matrix. (The Cholesky decomposition can be thought of as a special case of the LU-decomposition described in Sec. A.1 for a symmetric matrix.) Then Eq. (5.92) becomes

$$\|\mathbf{L}^{-1} \cdot (\boldsymbol{a}-\hat{\boldsymbol{a}})\| = \sqrt{\Delta\chi^2}. \tag{5.94}$$

To construct the ellipsoid, consider a set of points $\{\boldsymbol{u}\}$ on a unit sphere, $\|\boldsymbol{u}\| = 1$; then map these points to points on the error ellipsoid through

$$\boldsymbol{a} = \hat{\boldsymbol{a}} + \sqrt{\Delta\chi^2}\, \mathbf{L} \cdot \boldsymbol{u}. \tag{5.95}$$

Often we are interested in two-dimensional projections of the confidence ellipsoids into a plane spanned by two of the parameters, say a_j and a_k. The corresponding sub-matrix $\boldsymbol{\Sigma}_{[jk]}$ is

$$\boldsymbol{\Sigma}_{[jk]} = \begin{bmatrix} \sigma_j^2 & \sigma_{jk} \\ \sigma_{jk} & \sigma_k^2 \end{bmatrix} \tag{5.96}$$

and its Cholesky decomposition is

$$\mathbf{L}_{[jk]} = \begin{bmatrix} \sigma_j & 0 \\ r_{jk}\sigma_k & (1-r_{jk}^2)^{1/2}\sigma_k \end{bmatrix} \tag{5.97}$$

where

$$r_{jk} = \frac{\sigma_{jk}}{\sigma_j \sigma_k} \tag{5.98}$$

is the *correlation coefficient* between a_j and a_k. With $\boldsymbol{u} = [\cos\theta, \sin\theta]$ being a one-parameter family of points on a unit circle, the error ellipse on the a_j-a_k plane is then given parametrically by

$$\begin{aligned} a_j &= \hat{a}_j + \sqrt{\Delta\chi^2}\,\sigma_j \cos\theta \\ a_k &= \hat{a}_k + \sqrt{\Delta\chi^2}\,\sigma_k [r_{jk}\cos\theta + (1-r_{jk}^2)^{1/2}\sin\theta] \end{aligned} \qquad 0 \le \theta \le 2\pi. \tag{5.99}$$

Although the method such as LU-decomposition (or Cholesky decomposition) that is described in Sec. A.1 can be used to solve the normal equations, a more natural approach is the *singular value decomposition* (SVD) method. The singular value decomposition of a $N \times M$ real matrix $\mathbf{X}$ is

$$\mathbf{X} = \mathbf{U} \cdot \mathbf{W} \cdot \mathbf{V}^{\mathsf{T}} \tag{5.100}$$

where $\mathbf{U}$ is a $N \times N$ orthogonal matrix (i.e., a matrix in which $\mathbf{U}^{-1} = \mathbf{U}^{\mathsf{T}}$), $\mathbf{V}$ is a $M \times M$ orthogonal matrix, and $\mathbf{W}$ is $N \times M$ rectangular diagonal matrix with non-zero components are only on the main diagonal, and these components, $w_j = W_{jj}$, $j = 0, 1, 2, \ldots, M-1$, are all non-negative. The overdetermined system of equations given by $\mathbf{X} \cdot \hat{\boldsymbol{a}} = \boldsymbol{b}$ can then be "solved" as

$$\hat{\boldsymbol{a}} = \mathbf{V} \cdot \mathbf{W}^{+} \cdot \mathbf{U}^{\mathsf{T}} \cdot \boldsymbol{b} \tag{5.101}$$

where $\mathbf{W}^{+}$ is the *pseudoinverse* of the matrix $\mathbf{W}$, which is formed by taking the transpose of $\mathbf{W}$ and taking the reciprocal of all the diagonal elements so that $W^{+}_{jj} = 1/w_j$, $j = 0, 1, 2, \ldots, M-1$. (The pseudoinverse, also known as the *Moore-Penrose inverse* can be written $\mathbf{W}^{+} = (\mathbf{W}^{\mathsf{T}} \cdot \mathbf{W})^{-1} \cdot \mathbf{W}^{\mathsf{T}}$.) The covariance matrix $\mathbf{\Sigma}$ is

$$\mathbf{\Sigma} = \mathbf{V} \cdot (\mathbf{W}^{\mathsf{T}} \cdot \mathbf{W})^{-1} \cdot \mathbf{V}^{\mathsf{T}}. \tag{5.102}$$

The geometric interpretation is that the $M \times M$ diagonal square matrix $(\mathbf{W}^{\mathsf{T}} \cdot \mathbf{W})^{-1} = \mathrm{diag}\{1/w_j^2\}$ gives the lengths of the principal axes of the error ellipsiod, and the orthogonal matrix $\mathbf{V}$ is a rotation in the parameter space so that the columns of $\mathbf{V}$ correspond to the components of the principal axes.

The advantage to using the SVD method is to handle situations in which the data do not clearly distinguish between two of the basis functions, say $f_j(x)$ and $f_k(x)$. When this occurs, the matrix $\mathbf{X}^{\mathsf{T}} \cdot \mathbf{X}$ becomes singular and methods such as LU-decomposition will be unable to solve the linear system. In the case of a singular value decomposition, the pathology will manifest itself as one of the diagonal elements of $\mathbf{W}$ being zero (or very close to zero). The solution is the following: when computing $\mathbf{W}^{+} = \mathrm{diag}\{1/w_j\}$, set any diagonal element in which w_j is small to zero rather than $1/w_j$. By doing so, the particular combination of basis functions that is degenerate is effectively being given a very large uncertainty.

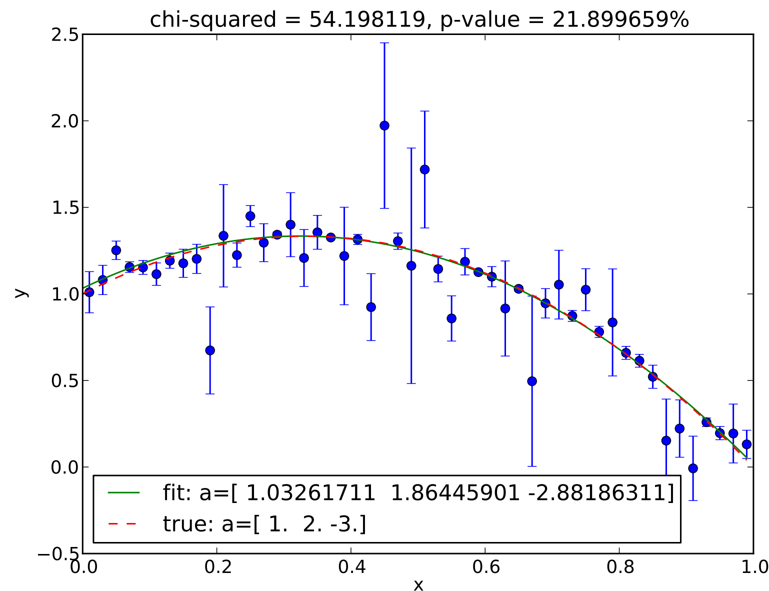

Figure 5.7: Results from running the program `svdfit.py` (Listing 5.8) for $N = 50$ data points fitted to a polynomial of degree 2 ($M = 3$) with coefficients $a_0 = 1$, $a_1 = 2$, and $a_3 = -3$, and random number seed 101. The resulting fit has $\hat{a}_0 = 1.03$, $\hat{a}_1 = 1.86$, $\hat{a}_2 = -2.88$, $\chi^2 = 54.2$, and p-value $p = 22\%$.

The program `svdfit.py` performs a polynomial fit to simulated data. The data points are generated as before but now with a true model given by a polynomial of specified order and coefficients. Figure 5.7 shows the results of the program `svdfit.py` for $N = 50$ data points following the quadratic function $y = a_0 + a_1 x + a_2 x^2$ with $a_0 = 1$, $a_1 = 1$, $a_2 = -3$ and a random number seed of 101; the estimated coefficients are $\hat{a}_0 = 1.03$, $\hat{a}_1 = 1.86$, and $\hat{a}_2 = -2.88$, and the fit has $\chi^2 = 54.2$ and a p-value of $p = 22\%$, which indicates that the fit is good. The 90% confidence error ellipses (1.64-sigma) for the parameters are shown in Fig. 5.8.

Listing 5.8: Program `svdfit.py`

```
1  import pylab, scipy.special, random, math
2
3  N = input('number of points -> ')
4  M = 1+input('polynomial order -> ')
5  a = pylab.zeros(M)
6  for j in range(M):
7      a[j] = input('polynomial coefficient a%d -> '%j)
8  seed = input('random seed -> ')
9  random.seed(seed)
10
```

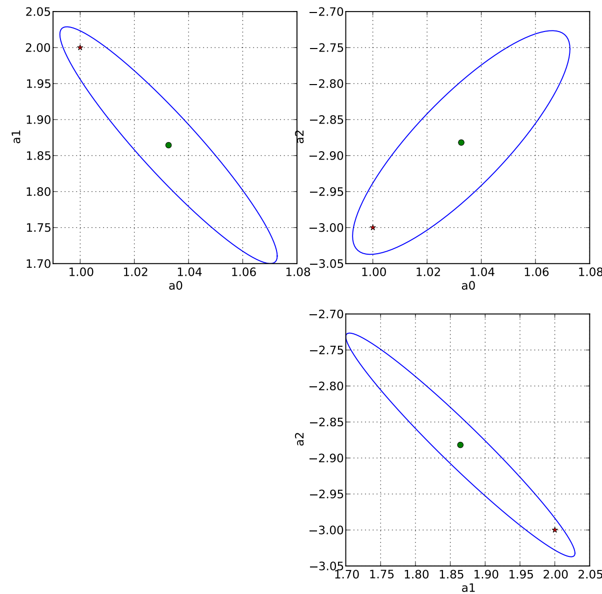

Figure 5.8: One-sigma (68% confidence) error ellipses for the measured parameters from running the program `svdfit.py` (Listing 5.8) with the same input parameters as in Fig. 5.7. The true values of the parameters are shown by a red star.

```
11  # generate random data with random uncertainties in y
12  x = pylab.arange(0.5/N, 1.0, 1.0/N)
13  dy = pylab.array([0.1*random.lognormvariate(0.0, 1.0) for i in range(N)
        ])
14  y = [sum(a[j]*x[i]**j for j in range(M)) for i in range(N)]
15  y = pylab.array([random.gauss(y[i], dy[i]) for i in range(N)])
16
17  # construct vector b and design matrix X
18  b = pylab.zeros(N)
19  X = pylab.zeros((N, M))
20  for i in range(N):
21      b[i] = y[i]/dy[i]
22      for j in range(M):
23          X[i,j] = x[i]**j/dy[i]
24
25  # compute fit parameters ahat and covariance matrix Sigma
26  (U, w, VT) = pylab.svd(X)
27  wmax = max(w)
28  Winv = pylab.zeros((M, N))
```

```
29 Sigma = pylab.zeros((M, M))
30 eps = 1e-6
31 for j in range(M):
32     if w[j] > eps*wmax:
33         Winv[j,j] = 1.0/w[j]
34     else:
35         Winv[j,j] = 0.0
36     Sigma[j,j] = Winv[j,j]**2
37 ahat = pylab.dot(VT.T, pylab.dot(Winv, pylab.dot(U.T, b)))
38 Sigma = pylab.dot(VT.T, pylab.dot(Sigma, VT))
39
40 # compute chi-square and p-value of the fit
41 chisq = pylab.norm(pylab.dot(X, ahat)-b)**2
42 p = 1.0-scipy.special.gammainc((N-M)/2.0, chisq/2.0)
43
44 # plot results
45 xfine = pylab.arange(0.0, 1.0, 0.01)
46 yfit = [sum(ahat[j]*xx**j for j in range(M)) for xx in xfine]
47 ytrue = [sum(a[j]*xx**j for j in range(M)) for xx in xfine]
48 pylab.errorbar(x, y, fmt='o', yerr=dy)
49 pylab.plot(xfine, yfit, label='fit: a='+str(ahat))
50 pylab.plot(xfine, ytrue, '--', label='true: a='+str(a))
51 pylab.xlabel('x')
52 pylab.ylabel('y')
53 pylab.title('chi-squared = %f, p-value = %f%%'%(chisq, 100.0*p))
54 pylab.legend(loc=3)
55
56 # plot 90% (1.64-sigma) error ellipses
57 cos = [math.cos(2.0*math.pi*i/100.0) for i in range(101)]
58 sin = [math.sin(2.0*math.pi*i/100.0) for i in range(101)]
59 pylab.figure(figsize=(10, 10))
60 for j in range(M-1):
61     for k in range(j+1, M):
62         pylab.subplot(M-1, M-1, (M-1)*j+k)
63         (s0, s1, s01) = (Sigma[j,j]**0.5, Sigma[k,k]**0.5, Sigma[j,k])
64         r = s01/(s0*s1)
65         s = (1.0-r**2)**0.5
66         ex = [ahat[j]+3.53**0.5*s0*cos[i] for i in range(len(cos))]
67         ey = [ahat[k]+3.53**0.5*s1*(r*cos[i]+s*sin[i]) for i in
68               range(len(cos))]
69         pylab.plot(ex, ey)
70         pylab.plot([ahat[j]], [ahat[k]], 'go')
71         pylab.plot([a[j]], [a[k]], 'r*')
72         pylab.xlabel('a%d'%j)
73         pylab.ylabel('a%d'%k)
74         pylab.grid()
75 pylab.show()
```

5.4 Bayesian inference

The statistical methods that have been described in the previous two sections have been *frequentist* in nature: the statements that have been made are all of the form "assuming the null-hypothesis, what is the probability of the observed data occurring." The use of the p-value to reject a null hypothesis and the use of confidence ellipsoids for fit parameters are methods of frequentist inference.

Bayesian inference, on the other hand, attempts to establish the probability of a hypothesis or a model given the observed data (in contrast to the frequentist approach of finding the probability of the data given a hypothesis or a model). Suppose that $P(D \mid H)$ is the *conditional probability* of observing the data D given a hypothesis H (this is what is used in frequentist inference). Then the conditional probability of the hypothesis given the data, which is known as the *posterior probability*, is given by Bayes's law,

$$P(H \mid D) = \frac{P(H)P(D \mid H)}{P(D)} \tag{5.103}$$

where $P(H)$ and $P(D)$ are unconditional probabilities: $P(H)$ is known as the *prior probability* of the hypothesis being true (the probability of the hypothesis before considering the data) and $P(D)$ is the known as the *evidence*, which is the total probability of the data under any hypothesis — it is essentially a normalization constant required to make $P(H \mid D)$ into a probability.

Bayes's law follows directly from the relationship between conditional probabilities and joint probabilities: If $P(A \cap B)$ is the joint probability of both A and B being true then then $P(A \cap B) = P(A \mid B)P(B)$ where $P(A \mid B)$ is the probability of A being true given that B is true and $P(B)$ is the unconditional probability of B being true. Also, $P(A \cap B) = P(B \mid A)P(A)$. By equating these two expressions for the joint probability $P(A \cap B)$ we arrive at Bayes's law, $P(B \mid A) = P(B)P(A \mid B)/P(A)$.

For the problem of data modelling, we wish to determine the posterior probability distribution of the parameters $\boldsymbol{a}$ given the observed data, which we will denote $\boldsymbol{x}$,

$$p(\boldsymbol{a} \mid \boldsymbol{x}) = \frac{p(\boldsymbol{a})L(\boldsymbol{a} \mid \boldsymbol{x})}{\int d\boldsymbol{a}\, p(\boldsymbol{a})L(\boldsymbol{a} \mid \boldsymbol{x})} \tag{5.104}$$

where $L(\boldsymbol{a} \mid \boldsymbol{x})$ is the *likelihood function* for the parameters, which is simply related to the probability density function of the data for those parameters $f(\boldsymbol{x}; \boldsymbol{a})$, by

$$L(\boldsymbol{a} \mid \boldsymbol{x}) = f(\boldsymbol{x}; \boldsymbol{a}). \tag{5.105}$$

The prior probability distribution for the parameters, $p(\boldsymbol{a})$, represents any previous knowledge we have on the parameter values, e.g., from previous experiments or from physical constraints. Notice that the denominator of the right hand side of Eq. (5.104) is the integral of the likelihood function over the entire parameter space, which is known as the *marginalized likelihood*; it the required normalization in order for $p(\boldsymbol{a} \mid \boldsymbol{x})$ to be a probability density function for the model parameters $\boldsymbol{a}$.

A method to locate the most probable parameter values $\hat{\boldsymbol{a}}$, i.e., the mode of the distribution $p(\boldsymbol{a} \mid \boldsymbol{x})$, is to perform a random walk in the parameter space, computing $p(\boldsymbol{a} \mid \boldsymbol{x})$ at every step. A *Markov chain Monte Carlo* (MCMC) method is one in which the random steps are taken so that the probability distribution of the parameter steps are given by the posterior distribution, $p(\boldsymbol{a} \mid \boldsymbol{x})$. In such a random walk, since the steps themselves map out the desired posterior distribution, we obtain not only the most likely value of the parameters but also their distribution.

We have already discussed the Metropolis algorithm for drawing random parameter values that follow a desired distribution $p(\boldsymbol{a} \mid \boldsymbol{x})$. Here we consider a generalization known as the *Metropolis-Hastings algorithm*. First we need to choose some *proposal distribution*, $q(\boldsymbol{a}'; \boldsymbol{a})$, that gives the probability density for moving to a new position $\boldsymbol{a}'$ from the current position $\boldsymbol{a}$. Suppose that at step n in our random walk we are point $\boldsymbol{a}_n$ in parameter space. Then, use the proposal distribution $q(\boldsymbol{a}'; \boldsymbol{a}_n)$ to choose some candidate point $\hat{a}'$ in the parameter space. Now construct the Hastings test ratio

$$\alpha(\boldsymbol{a}_n, \boldsymbol{a}') = \min\left\{1, \frac{p(\boldsymbol{a}' \mid \boldsymbol{x})}{p(\boldsymbol{a}_n \mid \boldsymbol{x})} \frac{q(\boldsymbol{a}_n; \boldsymbol{a}')}{q(\boldsymbol{a}'; \boldsymbol{a}_n)}\right\}. \tag{5.106}$$

This is the probability that we accept the proposed point $\boldsymbol{a}'$, so we generate a random number u uniformly distributed between 0 and 1 and if $u \le \alpha(\boldsymbol{a}_n, \boldsymbol{a}')$ then we accept the proposed move and set $\boldsymbol{a}_{n+1} = \boldsymbol{a}'$; otherwise, we reject the proposed move and stay at the current position by setting $\boldsymbol{a}_{n+1} = \boldsymbol{a}_n$.

The Metropolis-Hastings algorithm yields a transition probability $w(\boldsymbol{a} \to \boldsymbol{a}')$ from some point $\boldsymbol{a}$ in parameter space to some new point $\boldsymbol{a}'$ in parameter space that is

$$w(\boldsymbol{a} \to \boldsymbol{a}') = q(\boldsymbol{a}'; \boldsymbol{a})\, \alpha(\boldsymbol{a}, \boldsymbol{a}'). \tag{5.107}$$

If we multiply this by $p(\boldsymbol{a} \mid \boldsymbol{x})$ we find

$$\begin{aligned} p(\boldsymbol{a} \mid \boldsymbol{x}) w(\boldsymbol{a} \to \boldsymbol{a}') &= p(\boldsymbol{a} \mid \boldsymbol{x})\, q(\boldsymbol{a}'; \boldsymbol{a})\, \alpha(\boldsymbol{a}, \boldsymbol{a}') \\ &= \min\left\{p(\boldsymbol{a} \mid \boldsymbol{x})\, q(\boldsymbol{a}'; \boldsymbol{a}), p(\boldsymbol{a}' \mid \boldsymbol{x})\, q(\boldsymbol{a}; \boldsymbol{a}')\right\} \\ &= p(\boldsymbol{a}' \mid \boldsymbol{x})\, q(\boldsymbol{a}; \boldsymbol{a}') \min\left\{\frac{p(\boldsymbol{a} \mid \boldsymbol{x})}{p(\boldsymbol{a}' \mid \boldsymbol{x})} \frac{q(\boldsymbol{a}'; \boldsymbol{a})}{q(\boldsymbol{a}; \boldsymbol{a}')}, 1\right\} \\ &= p(\boldsymbol{a}' \mid \boldsymbol{x})\, q(\boldsymbol{a}; \boldsymbol{a}')\, \alpha(\boldsymbol{a}', \boldsymbol{a}) \\ &= p(\boldsymbol{a}' \mid \boldsymbol{x})\, w(\boldsymbol{a}' \to \boldsymbol{a}). \end{aligned} \tag{5.108}$$

This is the *detailed balance* equation that implies that the random walk will attain a physical equilibrium of reversible transitions between points. The probability distribution for landing at point $\boldsymbol{a}'$ is found by integrating the $p(\boldsymbol{a})\, w(\boldsymbol{a} \to \boldsymbol{a}')$ over all possible starting positions $\boldsymbol{a}$, which is

$$\int w(\boldsymbol{a} \to \boldsymbol{a}')\, p(\boldsymbol{a} \mid \boldsymbol{x})\, d\boldsymbol{a} = p(\boldsymbol{a}' \mid \boldsymbol{x}) \int w(\boldsymbol{a}' \to \boldsymbol{a})\, d\boldsymbol{a} = p(\boldsymbol{a}' \mid \boldsymbol{x}) \tag{5.109}$$

and so we see that the probability distribution for $\boldsymbol{a}'$ is indeed the desired distribution $p(\boldsymbol{a}' \mid \boldsymbol{x})$. Note that the $w(\boldsymbol{a}' \to \boldsymbol{a})$ is the probability of arriving at position $\boldsymbol{a}$

given that we started at position $\boldsymbol{a}'$, and so if we integrate it over all possible ending points $\boldsymbol{a}$ the result is the probability that we go *somewhere*, which is unity.

Notice the following important feature of the Markov chain Monte Carlo method: The posterior distribution (which we wish to determine) appears in the Hastings test ratio α only as a ratio. That is, in order to calculate α, we need only

$$p(\boldsymbol{a})\,L(\boldsymbol{a}\mid\boldsymbol{x}) \propto p(\boldsymbol{a}\mid\boldsymbol{x}), \tag{5.110}$$

and we do not need to know the normalization constant for the distribution $p(\boldsymbol{a}\mid\boldsymbol{x})$. The likelihood is a known function and the prior distribution $p(\boldsymbol{a})$ is also a known function.

Although the choice of the proposal distribution can be (almost) arbitrary, the effectiveness of the Markov chain Monte Carlo can depend sensitively on it. We want the proposal to be broad enough that we can explore the parameter space effectively should our initial guess at the parameters, $\boldsymbol{a}_0$, be far away from the most-probable values, $\hat{\boldsymbol{a}}$, otherwise it may take an arbitrarily large number of steps to find the region where the parameters are truly maximized. On the other hand, if the distribution is too broad then when we find ourselves in the right region of parameter space we may find ourselves rejecting nearly every proposal that takes us away from that region, which would prevent us from effectively exploring the parameter space in the high probability region.

A useful method for performing a Markov chain Monte Carlo is to begin with a burn-in time. Because our initial guess $\boldsymbol{a}_0$ might be in a very unlikely region of parameter space, it might take a number of steps before the chain moves to the correct region and begins to behave in some sense of equilibrium. During the burn-in period the proposal distribution should be relatively broad so that the chain can move about effectively until it finds the region of high likelihood. The width of the proposal distribution should be tuned during the burn-in period to achieve some reasonable rate of accepting proposals: if the *acceptance rate* is too low, i.e., if the fraction of proposals that are accepted is too small (say, $< 10\%$) then we should be taking smaller steps; however, if the acceptance rate is too high, i.e., if the fraction of proposals that are accepted is too large (say, $> 50\%$), then we should be taking bigger steps.

After the burn-in period, the chain is in equilibrium and explores the most likely region of parameter space, with the individual steps sampling from the posterior distribution that we wish to determine. The post-burn-in phase is then used to measure the posterior distribution.

An example will help to clarify the method. Suppose that there are N data points, $\{x_i\}$, $i = 0, 1, 2, \ldots, N-1$, that are Gaussian random deviates having a mean μ_1 for the first M points, $i < M$ (the first stage), and having a mean μ_2 for the remaining $N-M$ points, $i >= M$ (the second stage). Our goal is to compute the posterior probability distribution $p(\boldsymbol{a}\mid\boldsymbol{x})$ where $\boldsymbol{a} = [\mu_1, \mu_2, M]$ are our model parameters. The likelihood of a proposed set of parameters $\boldsymbol{a}$ is

$$L(\boldsymbol{a}\mid\boldsymbol{x}) = f(\boldsymbol{x};\boldsymbol{a}) = (2\pi\sigma^2)^{N/2}\exp\left[-\sum_{i=0}^{M-1}\frac{(x_i-\mu_1)^2}{2\sigma^2} - \sum_{i=M}^{N-1}\frac{(x_i-\mu_2)^2}{2\sigma^2}\right] \tag{5.111}$$

while we take our prior distributions to be uniform distributions with $p(\mu_1)$ and $p(\mu_2)$ being constant over the entire μ_1-μ_2 plane (this is known as an *improper prior* because it is not normalizable) and $p(M) = 1/N$ for $0 \le M \le N-1$ and $p(M) = 0$ otherwise.

We will adopt a proposal distribution that is a bivariate Gaussian distribution in μ_1 and μ_2,

$$q(\mu_1', \mu_2'; \mu_1, \mu_2) = \frac{1}{2\pi\sigma_\mu^2} \exp\left[-\frac{(\mu_1' - \mu_1)^2 + (\mu_2' - \mu_2)^2}{2\sigma_\mu^2}\right], \tag{5.112}$$

where σ_μ determines the step-size in the μ_1-μ_2 plane. Initially we will take a fairly large value $\sigma_\mu = 1$ but we will adjust this parameter during the burn-in phase so that we maintain a reasonable proposal acceptance rate. The proposal distribution for the transition point M, $q(M'; M)$, will be a uniform distribution of integers in the range $M - \Delta M$ to $M + \Delta M$ where ΔM is the typical step size in M. We begin with a step size of $\Delta M = N/10$ and we again adjust this value during the burn in phase.

Note that both the proposal distributions $q(\mu_1', \mu_2'; \mu_1, \mu_2)$ and $q(M'; M)$ are symmetric: $q(\mu_1', \mu_2'; \mu_1, \mu_2) = q(\mu_1, \mu_2; \mu_1', \mu_2')$ and $q(M'; M) = q(M; M')$, so the ratio of the proposal distributions in the Hastings test ratio α will be unity. That is, the Metropolis-Hastings algorithm reduces to the Metropolis algorithm when one chooses symmetric proposal distributions. The acceptance test ratio is therefore

$$\alpha(\boldsymbol{a}_n, \boldsymbol{a}') = \begin{cases} 0 & M' < 0 \text{ or } M' \ge N, \\ \Lambda & 0 \le M' < N \text{ and } \Lambda \le 1 \\ 1 & \text{otherwise} \end{cases} \tag{5.113}$$

(note the prior on M' gives rise to the condition $\alpha = 0$ when $M < 0$ or $M \ge N$) where Λ is the *likelihood ratio*,

$$\Lambda = \frac{L(\boldsymbol{a}' \mid \boldsymbol{x})}{L(\boldsymbol{a}_n \mid \boldsymbol{x})}. \tag{5.114}$$

It is more convenient to compute the logarithm of the likelihood ratio,

$$\begin{aligned} \ln \Lambda = &- \sum_{i=0}^{M'-1} \frac{(x_i - \mu_1')^2}{2\sigma^2} - \sum_{i=M'}^{N-1} \frac{(x_i - \mu_2')^2}{2\sigma^2} \\ &+ \sum_{i=0}^{M_n-1} \frac{(x_i - \mu_{1,n})^2}{2\sigma^2} + \sum_{i=M_n}^{N-1} \frac{(x_i - \mu_{2,n})^2}{2\sigma^2} \\ = &\frac{\mu_1' S_{M'-1}}{\sigma^2} + \frac{\mu_2'(S_{N-1} - S_{M'})}{\sigma^2} - \frac{M'\mu_1'^2 + (N - M')\mu_2'^2}{2\sigma^2} \\ &- \frac{\mu_{1,n} S_{M_n-1}}{\sigma^2} - \frac{\mu_{2,n}(S_{N-1} - S_{M_n})}{\sigma^2} + \frac{M_n\mu_{1,n}^2 + (N - M_n)\mu_{2,n}^2}{2\sigma^2} \end{aligned} \tag{5.115}$$

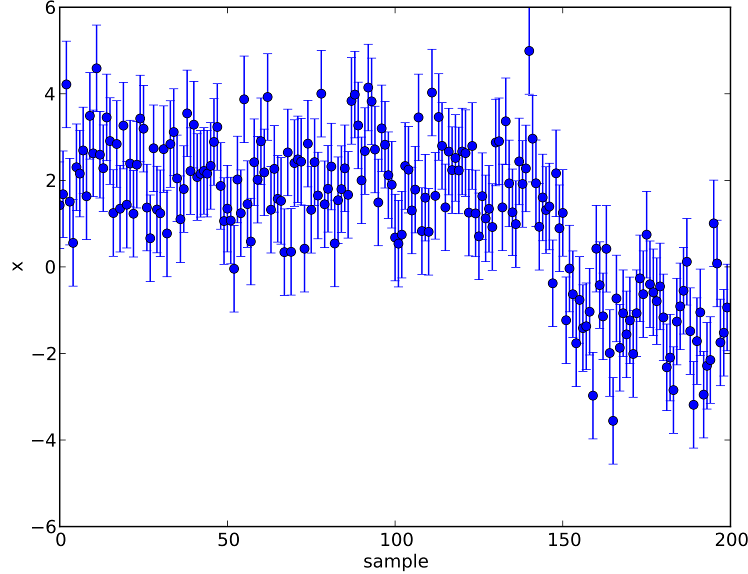

Figure 5.9: The Gaussian data generated by `mcmc.py` (Listing 5.9) has $N = 200$ data points; the first $M = 150$ of the data points have a mean of $\mu_1 = 2$ and a standard deviation $\sigma = 1$, while the remaining $N - M = 50$ data points have a mean of $\mu_2 = -1$ and a standard deviation $\sigma = 1$. A random seed of 101 was used to generate this data. The program `mcmc.py` then uses a Markov chain Monte Carlo method to estimate the values of μ_1, μ_2, and M.

where we can pre-compute the sums S_k, $k = 0, 1, 2, \ldots, N-1$,

$$S_k = \sum_{i=0}^{k} x_i. \tag{5.116}$$

Our initial guess for the parameters will be $\mu_1 = \mu_2 = 0$ and $M = N/2$.

The program `mcmc.py` performs a Markov chain Monte Carlo to estimate posterior probability distribution for the values μ_1, μ_2, and M. Figure 5.9 shows the data generated for $N = 200$ points with a first-stage mean of $\mu_1 = 2$, a second-stage mean of $\mu_2 = -1$, a transition from first- to second-stage at $M = 150$ points, and a random number seed of 101. We monitor the evolution of the estimated parameters in Fig. 5.10. The burn-in period is during the first 1000 points (it is in the red shaded region). During this time, the value of step size control parameters σ_μ and ΔM are adjusted to maintain an acceptance rate between 10% and 50%. At the end of the burn-in period, the chain has located the peak of the posterior distribution, and the subsequent measurement period (100 000 steps) are used to measure the posterior probability distribution, which is shown in Fig. 5.11. Notice

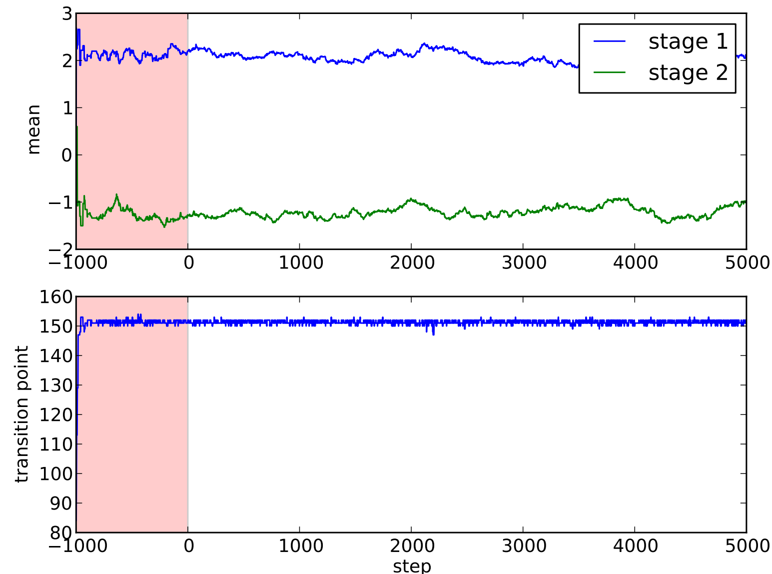

Figure 5.10: The evolution of the estimated parameter values μ_1, μ_2, and M as a function of Markov chain step performed by `mcmc.py` (Listing 5.9) for the data shown in Fig. 5.9. The first 1000 data points in the red shaded region are during the burn-in period; subsequently, the random steps are used as samples of the posterior probability. Only 5000 points following the burn-in period are shown.

from Fig. 5.10 that the correct value of M is determined quite accurately, and from Fig. 5.11 that the true value of μ_1 and μ_2, which is shown as a red star, is found close to the central region of the posterior probability distribution.

Listing 5.9: Program `mcmc.py`

```
import pylab, random, math, mpl_toolkits.mplot3d

# input parameters
N = input('number of points -> ')
Mtrue = input('transition point -> ')
mu1true = input('mean of first stage -> ')
mu2true = input('mean of second stage -> ')
seed = input('random seed -> ')
random.seed(seed)

# generate random data with fixed uncertainty
sigma = 1.0
x = [random.gauss(mu1true, sigma) for i in range(Mtrue)]
```

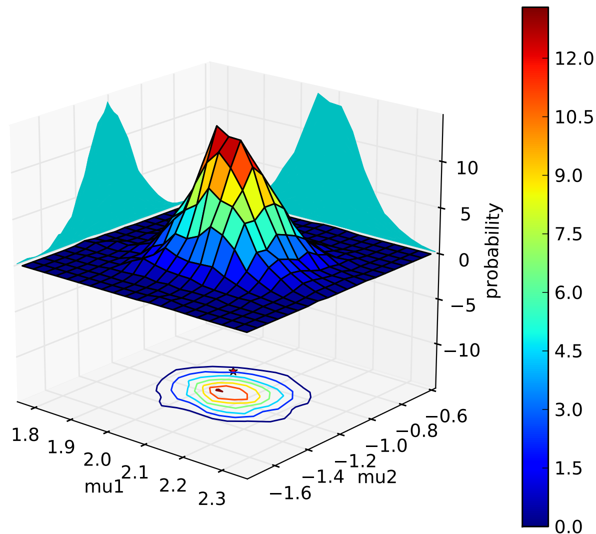

Figure 5.11: The posterior distribution for parameter values μ_1 and μ_2 found by `mcmc.py` (Listing 5.9) for the data shown in Fig. 5.9 (the posterior distribution for M is sharply peaked at the correct value $M = 150$). The red star indicates the location of the true values $\mu_1 = 2$ and $\mu_2 = -1$.

```
14 x += [random.gauss(mu2true, sigma) for i in range(Mtrue, N)]
15
16 # cumulative sum of x used to compute likelihoods
17 S = [sum(x[:i]) for i in range(N)]
18
19 # set up lists to store the steps
20 nburn = 1000   # number steps during burn−in phase
21 nmeas = 100000   # number of measurement steps after burn−in
22 M = [0]∗(nburn+nmeas)
23 mu1 = [0.0]∗(nburn+nmeas)
24 mu2 = [0.0]∗(nburn+nmeas)
25
26 # perform MCMC
27 dM = int(0.1∗N+0.5)   # step size for M
28 dmu = 1.0   # step size for mu1 and mu2
29 M[0] = N//2   # initial guess for transition point
30 accept = 0
31 for step in range(1, nburn+nmeas):
32     M[step] = M[step−1]+random.randint(−dM, dM)
33     mu1[step] = random.gauss(mu1[step−1], dmu)
```

```
34     mu2[step] = random.gauss(mu2[step-1], dmu)
35     logLambda = mu1[step]*S[M[step]-1]+mu2[step]*(S[N-1]-S[M[step]])
36     logLambda -= 0.5*(M[step]*mu1[step]**2+(N-M[step])*mu2[step]**2)
37     logLambda -= mu1[step-1]*S[M[step-1]-1]+mu2[step-1]*(S[N-1]-S[M[
        step-1]])
38     logLambda += 0.5*(M[step-1]*mu1[step-1]**2+(N-M[step-1])*mu2[step-
        1]**2)
39     logLambda /= sigma**2
40     if M[step] < 0 or M[step] >= N:  # reject step if M is out of bounds
41         logalpha = -float('inf')
42     else:
43         logalpha = min(0, logLambda)
44     if logalpha >= 0 or math.log(random.random()) < logalpha:  # accept
        step
45         accept += 1
46     else:
47           # reject step: reset parameters to previous values
48         M[step] = M[step-1]
49         mu1[step] = mu1[step-1]
50         mu2[step] = mu2[step-1]
51     # during burn-in phase, adjust step sizes to control rejection rate
52     if step < nburn and step%20 == 0:  # check every 20 steps
53         rate = accept/20.0  # acceptance rate
54         if rate < 0.1:  # rate too low: take smaller steps
55             dmu *= 0.5
56             dM = 1+int(dM//2)
57         if rate > 0.5:  # rate too high: take bigger steps
58             dmu *= 2.0
59             dM = 2*dM
60         accept = 0
61 
62 # plot data
63 pylab.figure()
64 pylab.errorbar(range(N), x, yerr=sigma, fmt='o')
65 pylab.ylabel('x')
66 pylab.xlabel('sample')
67 
68 # plot traces
69 nplot = 5000  # number of points after burn-in to plot
70 pylab.figure()
71 pylab.subplot(2, 1, 1)
72 pylab.plot(range(-nburn, nplot), mu1[:nburn+nplot], label='stage 1')
73 pylab.plot(range(-nburn, nplot), mu2[:nburn+nplot], label='stage 2')
74 pylab.ylabel('mean')
75 pylab.xlim(-nburn, nplot)
76 pylab.axvspan(-nburn, 0, facecolor='r', alpha=0.2)
77 pylab.legend()
78 pylab.subplot(2, 1, 2)
79 pylab.plot(range(-nburn, nplot), M[:nburn+nplot])
80 pylab.ylabel('transition point')
```

```
81  pylab.xlim(-nburn, nplot)
82  pylab.axvspan(-nburn, 0, facecolor='r', alpha=0.2)
83  pylab.xlabel('step')
84
85  # plot probability distribution in mu1-mu2 plane
86  fig = pylab.figure()
87  (Z, xedges, yedges) = pylab.histogram2d(mu1[nburn:], mu2[nburn:], bins
        =20,
88          normed=True)
89  X = 0.5*(xedges[:-1]+xedges[1:])  # centers of histogram bins
90  Y = 0.5*(yedges[:-1]+yedges[1:])  # centers of histogram bins
91  (Y, X) = pylab.meshgrid(Y, X)
92  axis = fig.gca(projection='3d', azim=-50, elev=20)
93  surf = axis.plot_surface(X, Y, Z, rstride=1, cstride=1, cmap=pylab.cm.
        jet)
94  axis.contour(X, Y, Z, zdir='z', offset=-pylab.amax(Z))
95  axis.contourf(X, Y, Z, 50, zdir='x', offset=min(mu1[nburn:]), colors='c'
        )
96  axis.contourf(X, Y, Z, 50, zdir='y', offset=max(mu2[nburn:]), colors='c'
        )
97  axis.set_xlabel('mu1')
98  axis.set_ylabel('mu2')
99  axis.set_zlabel('probability')
100 axis.set_zlim(-pylab.amax(Z), pylab.amax(Z))
101 axis.plot([mu1true], [mu2true], [-pylab.amax(Z)], 'r*')
102 fig.colorbar(surf)
103 pylab.show()
```

Appendix A

Algorithms

First: a warning! None of the routines presented here should be used in serious numerical work. These routines are intended to demonstrate various algorithms, but they are not written for efficiency or robustness. Almost certainly there will be packages that provide much better implementations than are given here, and these should be used instead. For example, in Python the module `numpy` (which is part of `pylab`) provides linear algebra routines that should be used rather than the illustrative ones given here. Similarly, `scipy` (also part of `pylab`) provides routines for finding roots, computing integrals, etc.

That said, I think it is helpful to learn about the algorithms in order to have a better understanding of what these packages are doing, so it is in this spirit that the implementations below are given.

A.1 Linear algebra

A common problem in linear algebra is the following: Given a $N \times N$ matrix $\mathbf{A}$, and a N-dimensional vector $\boldsymbol{b}$, solve the linear system

$$\mathbf{A} \cdot \boldsymbol{x} = \boldsymbol{b} \tag{A.1}$$

for the N-dimensional vector $\boldsymbol{x}$.

The approach that we will consider here, known as *LU decomposition*, was introduced by Alan Turing. The idea is to factor the matrix $\mathbf{A}$ into the product of a lower-triangular matrix $\mathbf{L}$ and a upper-triangular matrix $\mathbf{U}$, i.e.,

$$\mathbf{A} = \mathbf{L} \cdot \mathbf{U} \tag{A.2}$$

where, e.g., if $N = 4$, we have

$$\mathbf{L} = \begin{bmatrix} L_{00} & 0 & 0 & 0 \\ L_{10} & L_{11} & 0 & 0 \\ L_{20} & L_{21} & L_{22} & 0 \\ L_{30} & L_{31} & L_{32} & L_{33} \end{bmatrix} \quad \text{and} \quad \mathbf{U} = \begin{bmatrix} U_{00} & U_{01} & U_{02} & U_{33} \\ 0 & U_{11} & U_{12} & U_{13} \\ 0 & 0 & U_{22} & U_{23} \\ 0 & 0 & 0 & U_{33} \end{bmatrix}. \tag{A.3}$$

If we can achieve this, then we can recast our linear problem as follows:

$$\mathbf{A}\cdot\boldsymbol{x} = \mathbf{L}\cdot\mathbf{U}\cdot\boldsymbol{x} = \boldsymbol{b} \tag{A.4}$$

so if we let $\boldsymbol{y} = \mathbf{U}\cdot\boldsymbol{x}$ then we must solve the system

$$\mathbf{L}\cdot\boldsymbol{y} = \boldsymbol{b} \tag{A.5a}$$

$$\mathbf{U}\cdot\boldsymbol{x} = \boldsymbol{y}. \tag{A.5b}$$

We first solve Eq. (A.5a) for $\boldsymbol{y}$ and then solve Eq. (A.5b) to solve for $\boldsymbol{x}$. This is not so hard since $\mathbf{L}$ and $\mathbf{U}$ are triangular matrices: the system of equations that must be solved for $\boldsymbol{y}$ are

$$\begin{aligned}
L_{00}y_0 = b_0 \quad &\rightarrow \quad y_0 = b_0/L_{00}\\
L_{10}y_0 + L_{11}y_1 = b_1 \quad &\rightarrow \quad y_1 = (b_1 - L_{10}y_0)/L_{11}\\
L_{20}y_0 + L_{21}y_1 + L_{22}y_2 = b_2 \quad &\rightarrow \quad y_2 = (b_2 - L_{20}y_0 - L_{21}y_1)/L_{22}\\
\vdots \quad & \quad\quad \vdots
\end{aligned}$$

so if these are evaluated from the top down then you always have all the values of y_i that you need to compute the next one; similarly the system of equations that must be solved for $\boldsymbol{x}$ are

$$\begin{aligned}
U_{00}x_0 + U_{01}x_1 + U_{02}x_2 + \cdots + U_{0,N-2}x_{N-2} + U_{0,N-1}x_{N-1} &= y_0\\
U_{11}x_1 + U_{12}x_2 + \cdots + U_{1,N-2}x_{N-2} + U_{1,N-1}x_{N-1} &= y_1\\
&\vdots\\
U_{N-2,N-2}x_{N-2} + U_{N-2,N-1}x_{N-1} &= y_{N-2}\\
U_{N-1,N-1}x_{N-1} &= y_{N-1}
\end{aligned}$$

which are then solved *from the bottom up* so that

$$\begin{aligned}
x_{N-1} &= y_{N-1}/U_{N-1,N-1}\\
x_{N-2} &= (y_{N-2} - U_{N-2,N-1}x_{N-1})/U_{N-2,N-2}\\
x_{N-3} &= (y_{N-3} - U_{N-3,N-1}x_{N-1} - U_{N-3,N-2}x_{N-2})/U_{N-3,N-3}\\
&\vdots
\end{aligned}$$

and, again, when solved in this order, all the values of x_i that are required at each step will have already been computed. Therefore the procedure is as follows: first construct the components y_i

$$y_i = \frac{1}{L_{ii}}\left(b_i - \sum_{j=0}^{i-1} L_{ij}y_j\right) \quad \text{for } i = 0, 1, \ldots, N-1 \tag{A.6a}$$

and then construct the components x_i

$$x_i = \frac{1}{U_{ii}}\left(y_i - \sum_{j=i+1}^{N-1} U_{ij}x_j\right) \quad \text{for } i = N-1, N-2, \ldots, 0. \tag{A.6b}$$

Again, notice that by counting i upward in computing the y_i values and downward in computing the x_i values, the needed values in the expressions are always available.

The challenge now is to do the factorization of $\mathbf{A}$ into $\mathbf{L} \cdot \mathbf{U}$. This requires us to solve a system of equations that has the form

$$\begin{array}{llll}
A_{00} = L_{00}U_{00} & A_{01} = L_{00}U_{01} & A_{02} = L_{00}U_{02} & \cdots \\
A_{10} = L_{10}U_{00} & A_{11} = L_{10}U_{01} + L_{11}U_{11} & A_{12} = L_{10}U_{02} + L_{11}U_{12} & \cdots \\
A_{20} = L_{20}U_{00} & A_{21} = L_{20}U_{01} + L_{21}U_{11} & A_{22} = L_{20}U_{02} + L_{21}U_{12} + L_{22}U_{22} & \cdots \\
\vdots & \vdots & \vdots & \ddots
\end{array}$$

which are N^2 equations for $N(N+1)$ unknowns. The extra N unknowns can be specified by setting the diagonal elements $L_{ii} = 1$. The equations can be arranged in column-major-order starting with the leftmost column as follows: for the first column we have

$$\begin{array}{lcl}
A_{00} = U_{00} & \rightarrow & U_{00} = A_{00} \\
A_{10} = L_{10}U_{00} & \rightarrow & L_{10} = A_{10}/U_{00} \\
A_{20} = L_{20}U_{00} & \rightarrow & L_{20} = A_{20}/U_{00} \\
A_{30} = L_{30}U_{00} & \rightarrow & L_{30} = A_{30}/U_{00} \\
\vdots & & \vdots
\end{array}$$

for the second column we have

$$\begin{array}{lcl}
A_{01} = U_{01} & \rightarrow & U_{01} = A_{01} \\
A_{11} = L_{10}U_{01} + U_{11} & \rightarrow & U_{11} = A_{11} - L_{10}U_{01} \\
A_{21} = L_{20}U_{01} + L_{21}U_{11} & \rightarrow & L_{21} = (A_{21} - L_{20}U_{01})/U_{11} \\
A_{31} = L_{30}U_{01} + L_{31}U_{11} & \rightarrow & L_{31} = (A_{31} - L_{30}U_{01})/U_{11} \\
\vdots & & \vdots
\end{array}$$

for the third column we have

$$\begin{array}{lcl}
A_{02} = U_{02} & \rightarrow & U_{02} = A_{02} \\
A_{12} = L_{10}U_{02} + U_{12} & \rightarrow & U_{12} = A_{12} - L_{10}U_{02} \\
A_{22} = L_{20}U_{02} + L_{21}U_{12} + U_{22} & \rightarrow & U_{22} = A_{22} - L_{20}U_{02} - L_{21}U_{12} \\
A_{32} = L_{30}U_{02} + L_{31}U_{12} + L_{32}U_{22} & \rightarrow & L_{32} = (A_{32} - L_{30}U_{02} - L_{31}U_{12})/U_{22} \\
\vdots & & \vdots
\end{array}$$

and so on. The equations can then be solved in the order indicated: for each $j =$

$0, 1, \ldots, N-1$, compute first

$$U_{ij} = \begin{cases} A_{ij} - \sum_{k=0}^{i-1} L_{ik} U_{kj} & \text{for } i = 0, 1, \ldots, j \\ 0 & \text{for } i = j+1, \ldots, N-1 \end{cases} \tag{A.7a}$$

and then compute

$$L_{ij} = \begin{cases} 0 & \text{for } i = 0, 1, \ldots, j-1 \\ 1 & \text{for } i = j \\ \dfrac{1}{U_{jj}} \left(A_{ij} - \sum_{k=0}^{j-1} L_{ik} U_{kj} \right) & \text{for } i = j+1, \ldots, N-1. \end{cases} \tag{A.7b}$$

If these procedures are performed in order before j is advanced then again all the components are calculated before they are required. This method is known as *Crout's algorithm*.

I should note that this method is not very numerically stable without "pivoting", which involves exchanging rows or columns so that you never end up dividing by zero in Eq. (A.7b). See the warning at the beginning of this appendix!

Once in LU-decomposed form, it is simple to compute the determinant of the matrix, $\det \mathbf{A} = \prod_{i=0}^{N-1} U_{ii}$, and the inverse can also be computed by solving the system

$$\mathbf{L} \cdot \mathbf{Y} = \mathbf{I} \tag{A.8a}$$

$$\mathbf{U} \cdot \mathbf{A}^{-1} = \mathbf{Y} \tag{A.8b}$$

which is akin to solving the system Eqs. (A.5) repeatedly (column-by-column) for the vectors $\boldsymbol{b}$ corresponding to each column of the identity matrix and the resulting $\boldsymbol{x}$ is then the corresponding column of the inverse matrix.

A special case of LU-decomposition is the case of a *tridiagonal matrix*, for which the only non-zero components lie along the diagonal and next to the diagonal. Suppose we wish to solve the equation

$$\begin{bmatrix} \beta_0 & \gamma_0 & 0 & 0 & 0 & \cdots & 0 & 0 \\ \alpha_1 & \beta_1 & \gamma_1 & 0 & 0 & \cdots & 0 & 0 \\ 0 & \alpha_2 & \beta_2 & \gamma_2 & 0 & \cdots & 0 & 0 \\ 0 & 0 & \alpha_3 & \beta_3 & \gamma_3 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \ddots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & \alpha_{N-3} & \beta_{N-3} & \gamma_{N-3} & 0 \\ 0 & 0 & 0 & 0 & \cdots & \alpha_{N-2} & \beta_{N-2} & \gamma_{N-2} \\ 0 & 0 & 0 & 0 & 0 & \cdots & \alpha_{N-1} & \beta_{N-1} \end{bmatrix} \cdot \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_{N-3} \\ x_{N-2} \\ x_{N-1} \end{bmatrix} = \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_{N-3} \\ b_{N-2} \\ b_{N-1} \end{bmatrix} \tag{A.9}$$

for the vector $\boldsymbol{x}$. Rather than representing the whole $N \times N$ matrix, we have only the 3 N-vectors $\boldsymbol{\alpha}$, $\boldsymbol{\beta}$, and $\boldsymbol{\gamma}$ (and we do not even need the components α_0 and γ_{N-1} of

these vectors). The solution can be achieved with a single forward decomposition and substitution followed by a back-substitution. The forward sweep is given by

$$y_i = \begin{cases} \dfrac{\gamma_i}{\beta_i} & \text{for } i = 0 \\ \dfrac{\gamma_i}{\beta_i - \alpha_i y_{i-1}} & \text{for } i = 1, 2, \ldots N-1 \end{cases} \tag{A.10a}$$

and

$$x_i = \begin{cases} \dfrac{b_i}{\beta_i} & \text{for } i = 0 \\ \dfrac{b_i - \alpha_i x_{i-1}}{\beta_i - \alpha_i y_{i-1}} & \text{for } i = 1, 2, \ldots N-1. \end{cases} \tag{A.10b}$$

The back-substitution is then

$$x_i = x_i - x_{i+1} y_i \qquad i = N-2, N-3, \ldots, 0. \tag{A.10c}$$

Example routines for performing LU-decomposition, `ludecomp`, and for using such a decomposition to compute the determinant of a matrix, `determinant`, solving a linear system, `solve`, finding the inverse of a matrix, `inverse`, and solving a tridiagonal linear system, `tridiag`, are given in the listing for module `linalg.py`.

Listing A.1: Module `linalg.py`

```
1  import pylab
2
3
4  def ludecomp(A):
5      """ Use Crout’s algorithm to perform LU decomposition of A. """
6
7      n = len(A)
8      L = pylab.zeros(A.shape)
9      U = pylab.zeros(A.shape)
10     for i in range(n):
11         L[i,i] = 1.0
12     for j in range(n):
13         for i in range(j+1):
14             U[i,j] = A[i,j]
15             for k in range(i):
16                 U[i,j] -= L[i,k]*U[k,j]
17         for i in range(j+1, n):
18             L[i,j] = A[i,j]
19             for k in range(j):
20                 L[i,j] -= L[i,k]*U[k,j]
21             L[i,j] /= U[j,j]
22     return (L, U)
23
24
25 def determinant(A):
```

```
26      """ Computes the determinant of a matrix. """
27
28      (L, U) = ludecomp(A)
29      det = U[0,0]
30      for i in range(1, len(A)):
31          det *= U[i,i]
32      return det
33
34
35  def solve(A, b):
36      """ Solves the linear system A.x = b for x. """
37
38      n = len(A)
39      (L, U) = ludecomp(A)
40      x = pylab.zeros(b.shape)
41      y = pylab.zeros(b.shape)
42      # forward substitute to solve equation L.y = b for y
43      for i in range(n):
44          y[i] = b[i]
45          for j in range(i):
46              y[i] -= L[i,j]*y[j]
47          y[i] /= L[i,i]
48      # back substitute to solve equation U.x = y for x
49      for i in reversed(range(n)):
50          x[i] = y[i]
51          for j in range(i+1, n):
52              x[i] -= U[i,j]*x[j]
53          x[i] /= U[i,i]
54      return x
55
56
57  def inverse(A):
58      """ Finds the inverse of A. """
59
60      # note that the routine solve works even if b is a matrix!
61      B = pylab.eye(len(A))  # the identity matrix
62      return solve(A, B)
63
64
65  def tridiag(alp, bet, gam, b):
66      """ Solves the linear system A.x = b for x where A is a tridiagonal
67      matrix with subdiagonal elements given in the vector alp, diagonal
68      elements given in the vector bet, and superdiagonal elements given
         in
69      the vector gam. """
70
71      n = len(bet)
72      x = pylab.zeros(b.shape)
73      y = pylab.zeros(b.shape)
74      y[0] = gam[0]/bet[0]
```

```
75      x[0] = b[0]/bet[0]
76      for i in range(1, n):
77          den = bet[i]-alp[i]*y[i-1]
78          y[i] = gam[i]/den
79          x[i] = (b[i]-alp[i]*x[i-1])/den
80      for i in reversed(range(n-1)):
81          x[i] -= x[i+1]*y[i]
82      return x
```

A.2 Root finding

The task of root finding is simply: given a function $f(x)$, find the value of x for which

$$f(x) = 0. \tag{A.11}$$

If we try to solve N equations with N unknowns, we have a multidimensional root finding problem

$$\boldsymbol{f}(\boldsymbol{x}) = 0. \tag{A.12}$$

We first consider the one-dimensional case.

The first task is to bracket the root of interest. By *bracket* we mean that we wish to find some interval $a < x < b$ in which the root exists. Furthermore, we require $\operatorname{sgn} f(a) = -\operatorname{sgn} f(b)$.

Once the root is bracketed, there are several methods that can be used to locate its value. The *bisection method*, which was described earlier, is a robust method in which the interval is continually bisected so that the root is contained in smaller and smaller intervals. If the desired accuracy of the root is ϵ then the number of iterations required to find the root is $\log_2(|b-a|/\epsilon)$.

A more powerful method is known as *Newton's method*. Suppose that our initial guess for the root is the value x_0. Then the Taylor series of our function about this guess is

$$f(x) = f(x_0) + f'(x_0)(x - x_0) + \cdots \tag{A.13}$$

and since we are looking for the root, $f(x) = 0$, we solve this equation for x to obtain our next guess,

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}. \tag{A.14}$$

This process is repeated iteratively with

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}. \tag{A.15}$$

This procedure has *quadratic* convergence, meaning that if the error at step i is ϵ_i then the error at step $i+1$ is $\epsilon_{i+1} \propto \epsilon_i^2$. To see this, note that

$$0 = f(x) = f(x_i) + f'(x_i)(x - x_i) + \frac{1}{2} f''(\xi_i)(x - x_i)^2 \tag{A.16}$$

where ξ_i is some number between x and x_i. Using Eq. (A.15) we see

$$x - x_{i+1} = \frac{1}{2}\frac{f''(\xi_i)}{f'(x_i)}(x - x_i)^2 \tag{A.17}$$

and therefore we have

$$\epsilon_{i+1} = \frac{1}{2}\frac{|f''(\xi_i)|}{|f'(x_i)|}\epsilon_i^2 \tag{A.18}$$

where $\epsilon_{i+1} = |x - x_{i+1}|$ and $\epsilon_i = |x - x_i|$.

Note that Newton's method requires not only the function $f(x)$ but also its derivative $f'(x)$. However, it is often the case that we do not know this derivative function. In such cases, it is possible to use the numerical difference approximation to the derivative,

$$f'(x) \approx \frac{f(x + dx) - f(x)}{dx}. \tag{A.19}$$

Example root solving routines by bisection, `bisect`, or by Newton's method, `newton`, are given in the listing for module `root.py`. The routine `bisect` performs a bisection search for a root given a bracketing interval. The routine `newton` uses the Newton method to compute the root of a function given an initial guess. If the user supplies a derivative function `dfdx` then that is used; otherwise the difference approximation is used.

Listing A.2: Module `root.py`

```
def bisect(f, a, b, eps=1e-6):
    """ Finds the root of function f in the interval (a,b) by bisection.
    """

    # orient the search so that f(a) < 0 and f(b) > 0
    if f(a) > 0:   # swap a and b
        (a, b) = (b, a)
    while abs(a-b) > eps:
        xmid = (a+b)/2.0
        if f(xmid) < 0:
            a = xmid
        else:
            b = xmid
    return xmid


def newton(f, x, dfdx=None, eps=1e-6):
    """ Finds the root of a function using Newton's method. """

    if dfdx is None:   # for estimating derivative
        delta = eps**0.5
    while True:
        fx = f(x)
        if dfdx is None:
```

```
24            dx = delta*x
25            if abs(dx) < delta:
26                dx = delta
27            df = (f(x+dx)-fx)/dx
28        else:
29            df = dfdx(x)
30        dx = -fx/df
31        x += dx
32        if abs(dx) < eps:
33            return x
```

Exercise A.1 A difficulty with Newton's method is that the iteration will not generically converge on the root: it is quite possible that at some iteration the guess may find itself at a point where $f'(x_i)$ is very close to zero; if this happens, the next iteration will shoot off. Consider the complex equation

$$z^3 - 1 = 0 \tag{A.20}$$

for which Newton's method gives the iteration

$$z_{i+1} = \tfrac{2}{3} z_i - \tfrac{1}{3} z_i^{-2}. \tag{A.21}$$

Explore the basin of convergence for the root $z = 1$, i.e., the region in the complex plane for which the initial guess eventually converges on the root.

Newton's method can be used to obtain recurrence relations for computing various functions. For example, the square root function, $\sqrt{y}$, can be obtained by solving for the root of the equation

$$x^2 - y = 0. \tag{A.22}$$

Here, $f(x) = x^2 - y$ so Newton's method yields the recurrence relation

$$x_{i+1} = \frac{1}{2}\left(x_i + \frac{y}{x_i}\right) \tag{A.23}$$

with $x_0 = 1$. Another example is to compute the reciprocal, $1/y$, of a number. Here we let $f(x) = y - 1/x$ and Newton's method then results in the recurrence relation

$$x_{i+1} = x_i(2 - y x_i). \tag{A.24}$$

This method works when $\frac{1}{2} \le y < 1$. Note: only multiplication and addition is required to do division.

Now consider the problem of multidimensional root finding. We wish to solve a system of N equations,

$$\boldsymbol{f}(\boldsymbol{x}) = 0 \tag{A.25}$$

for the N unknowns, $\boldsymbol{x}$. The Taylor expansion of $\boldsymbol{f}(\boldsymbol{x}_i + \Delta\boldsymbol{x}_i)$ is

$$\boldsymbol{f}(\boldsymbol{x}_i + \Delta\boldsymbol{x}_i) = \boldsymbol{f}(\boldsymbol{x}_i) + \mathbf{J}(\boldsymbol{x}_i)\cdot\Delta\boldsymbol{x}_i + O(\Delta\boldsymbol{x}_i^2) \tag{A.26}$$

where $\mathbf{J}$ is the Jacobian matrix

$$\mathbf{J} = \begin{bmatrix} \partial f_1/\partial x_1 & \cdots & \partial f_1/\partial x_N \\ \vdots & \ddots & \vdots \\ \partial f_N/\partial x_1 & \cdots & \partial f_N/\partial x_N \end{bmatrix}. \tag{A.27}$$

We thus determine $\Delta\boldsymbol{x}_i$ by solving the matrix equation

$$\mathbf{J}(\boldsymbol{x}_i)\cdot\Delta\boldsymbol{x}_i = -\boldsymbol{f}(\boldsymbol{x}_i) \tag{A.28}$$

and iterate our guess

$$\boldsymbol{x}_{i+1} = \boldsymbol{x}_i + \Delta\boldsymbol{x}_i. \tag{A.29}$$

If the Jacobian is not known, it can be approximated with a finite difference approximation.

A.3 Minimization

Finding the minimum of a function $f(x)$ is somewhat different from finding the roots of the function, but there are some similarities. The first task is to bracket the minimum. However, here a bracket is a triplet of points (a,b,c) with $a < b < c$ and where $f(b) < f(a)$ and $f(b) < f(c)$. If this is the case then we know that the minimum (or a minimum) of $f(x)$ must lie somewhere between a and c.

The analog of the bisection method of root finding is the following: Suppose that the interval (a,b) is larger than the interval (b,c); then choose a trial value x somewhere in the interval between a and b and compute $f(x)$. If $f(x) < f(b)$ then our new bracketing triplet is (a,x,b) but if $f(x) > f(b)$ then our new bracketing triplet is (x,b,c).

Ideally we would like the width of the bracket to be the same for either of these outcomes, so we require $b-a = c-x$. Now we need to consider where the value of b should have been placed initially. Suppose that the distance $b-a$ is a factor φ larger than the distance $c-b$. Let $\Delta = c-b = x-a$ so that $b-a = \varphi\Delta = c-x$. We want the new bracketing triplet to have the same proportion. In the case that the new bracketing triplet is (a,x,b), then, we would like $x-a = \varphi(b-x)$. However, $x-a = \Delta$ and $b-x = (c-x)-(c-b) = \varphi\Delta - \Delta$. We therefore find $\Delta = \varphi^2\Delta - \varphi\Delta$ or

$$\varphi^2 - \varphi - 1 = 0. \tag{A.30}$$

The (positive) solution to this equation is

$$\varphi = \frac{1+\sqrt{5}}{2} \approx 1.61803399 \tag{A.31}$$

which is known as the *golden ratio*.

With each iteration, the bracketing interval will be reduced to a fraction $1/\varphi \approx 0.618033989$ of its previous size — almost as good as the fraction 0.5 that was achieved in each iteration in the bisection search for a root.

Because we are searching for the *minimum* of a function, we might not be able to achieve the expected accuracy in the location of the minimum. This is because a function is relatively insensitive to offsets Δx from the position of the minimum. Note that if x is the minimum of the function then

$$\begin{aligned} f(x+\Delta x) &\approx f(x) + \frac{1}{2} f''(x)\,\Delta x^2 \\ &= f(x)(1 \pm \epsilon) \end{aligned} \tag{A.32}$$

where

$$\epsilon = \frac{|x^2 f''(x)|}{2|f(x)|}\left(\frac{\Delta x}{x}\right)^2. \tag{A.33}$$

If $|x^2 f''(x)| \sim |f(x)|$ then we see that the tolerance that we should strive for is

$$|\Delta x| \sim |x|\sqrt{\epsilon}. \tag{A.34}$$

The minimization method discussed above is only applicable to one-dimensional problems, and it will only find the minimum within the given bracket, which might be a local minimum rather than a global minimum. Finding a global minimum of a (possibly multidimensional) function is a more complex problem. A useful, stochastic method for searching for the approximate global minimum is *simulated annealing*. The method is *not* guaranteed to find the true global minimum of the function (to within some tolerance), but it is useful in finding an acceptable minimum, particularly in problems of optimization.

Annealing is a process in which a metal is brought to high temperatures and then cooled, which allows the collection of atoms to settle into a minimum energy state. In case of numerical minimization of a function, points within the domain of the function are selected randomly, and the current guess of the minimum is updated probabilistically. The domain is explored widely at first, where transitions from one point in the domain to another are frequently accepted, even if the value of the function increases in doing so by borrowing energy from the reservoir. At temperature T the probability of being in a state with energy E is proportional to $e^{-E/k_B T}$ (the Boltzmann distribution) where k_B is Boltzmann's constant. As cooling occurs, the exploration narrows as transitions to points with larger values of the function are accepted less and less often when they involve an increase in energy. A temperature parameter controls the degree to which unfavorable transitions will be allowed. A cooling schedule describes how the temperature decreases with iteration. At very low temperature, the system settles down to the local minimum, which is hopefully quite close to the global minimum, if the cooling takes place slowly enough. (If the system is cooled to $T = 0$ immediately, only the local minimum in the vicinity of the initial state will be found.)

Suppose the current guess of the position of the minimum of some function $f(\boldsymbol{x})$ is $\boldsymbol{x}_0$. A new position $\boldsymbol{x}$ is proposed. If the current temperature parameter is T then this proposed position is accepted with probability

$$P = \mathrm{e}^{-[f(\boldsymbol{x})-f(\boldsymbol{x}_0)]/T} \tag{A.35}$$

if $f(\boldsymbol{x}) > f(\boldsymbol{x}_0)$; otherwise the proposal is always accepted. This is the Metropolis algorithm discussed in Sec. 4.2. By sometimes allowing such uphill moves, the annealing process allows wider exploration of the parameter space and is less susceptible to being trapped in a local minimum. As T is decreased, such uphill moves become less frequent, and the annealing process becomes more and more greedy, demanding that only downhill moves be allowed.

The initial temperature T should be chosen to be larger than the range of the function over its domain so that the entire domain is explored at high temperature. (In the implementation below, if an initial temperature is not specified, twice the value of $f(\boldsymbol{x}_0)$ is used where $\boldsymbol{x}_0$ is the initial guess.) At any given temperature, several proposals are considered before reducing the temperature (100 times the number of dimensions in $\boldsymbol{x}$ in the implementation below), after which the temperature is lowered by 5%. Eventually, when the temperature is low and new proposals are always rejected, the current position is taken to be the minimum.

The listing for `minimize.py` contains three routines: the first one, `bracket`, is designed to search for a bracketing triplet (a, b, c) by walking downhill from an initial guessed interval (a, b). The second routine, `golden`, employs the golden section search algorithm to find a minimum given a bracketing triplet (a, b, c). The third routine, `anneal`, performs simulated annealing to find an approximate global minimum.

Listing A.3: Module `minimize.py`

```
1  import math, random
2
3  goldenratio = 0.5+0.5*5.0**0.5
4
5
6  def bracket(f, a, b):
7      """ Brackets the minimum of a function. """
8
9      fa = f(a)
10     fb = f(b)
11     if fb > fa:  # swap a and b so that f(b) < f(a)
12         (a, b) = (b, a)
13         (fa, fb) = (fb, fa)
14     c = b+(b-a)*goldenratio
15     fc = f(c)
16     while fb > fc:  # keep going downhill
17         (a, b) = (b, c)
18         c = b+(b-a)*goldenratio
19         (fb, fc) = (fc, f(c))
```

```
20      return (a, b, c)
21
22
23  def golden(f, a, b, c, eps=1e-6):
24      """ Uses a golden section search for the minimum of a function. """
25
26      tolerance = eps**0.5
27      fb = f(b)
28      while abs(a-c) > tolerance*(abs(a)+abs(c)):
29          # make sure that b is closer to c
30          if abs(b-a) < abs(c-b):  # swap a with c
31              (a, c) = (c, a)
32          x = a+(b-a)/goldenratio
33          fx = f(x)
34          if fx < fb:
35              (a, b, c) = (a, x, b)
36              fb = fx
37          else:
38              (a, b, c) = (x, b, c)
39      if fx < fb:
40          return (x, fx)
41      else:
42          return (b, fb)
43
44
45  def anneal(f, x0, xnew, T0=None):
46      """ Uses simulated annealing to find the minimum of a function. """
47
48      # get number of dimensions of x0, or 1 if scalar
49      try: ndim = len(x0)
50      except: ndim = 1
51
52      f0 = f(x0)
53      T = T0 if T0 is not None else 2.0*f0
54      anneal.state = (T, x0, f0) # save state for possible external access
55
56      # loop until no step accepted at current temperature
57      accept = True
58      while accept:
59          accept = False
60          for i in range(100*ndim): # steps at this temperature
61              x = xnew(x0) # get proposed new position
62              fx = f(x)
63              if fx < f0 or random.random() < math.exp((f0-fx)/T):
64                  accept = True # the step has been accepted
65                  (x0, f0) = (x, fx) # update position
66                  anneal.state = (T, x0, f0) # update state
67          T *= 0.95 # exponential cooling
68      return (x0, f0)
```

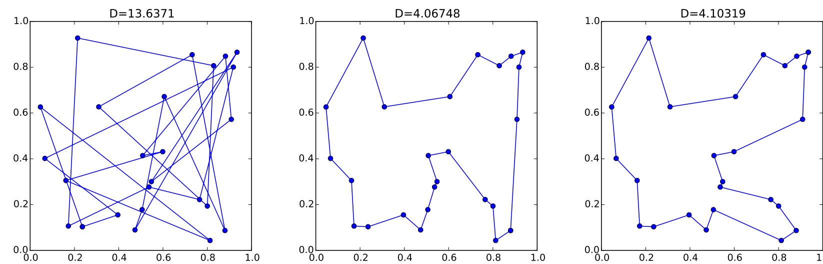

Figure A.1: Left panel: the initial route of the salesperson. The distance of this path is 13.637. Middle panel: the optimized solution found by simulated annealing with a random number seed of 1 used in the simulation. The distance of this solution is 4.067. Right panel: the optimized solution found by simulated annealing with a random number seed of 2 used in the simulation. The distance of this solution is 4.103.

An illustration of simulated annealing is in solving the traveling salesperson problem: given N towns, what route should a salesperson take in order to minimize the total distance that they must travel while visiting each town and returning to the original town?

The program `traveler.py` uses simulated annealing to attempt to solve this problem for $N = 25$ towns. First, the town positions are chosen randomly (with a fixed random number seed so that the same town locations are used for different simulations). The order of the towns in the list of town positions is taken to be the order that they are visited by the salesperson. A function `distance` computes the distance the salesperson must travel to visit all the towns in order, and return to the starting town. To generate a new path, two towns are chosen at random and are swapped in the routine `townswap`. These routines and the initial list of towns are passed to the routine `anneal`. The results of two simulations, with random number seeds of 1 and 2, yield paths with a total distance of 4.067 and 4.103 respectively, while the initial path had distance 13.637. These are shown in Fig. A.1. While simulated annealing does not always result in the global minimum, both of the solutions are close to the optimal solution.

Listing A.4: Program `traveler.py`

```
import pylab, random, minimize

# (x,y) coordinates for N random town locations
random.seed(4)  # always use the same town locations
N = 25
p = [(random.random(), random.random()) for i in range(N)]

# get random seed for this simulated annealing
seed = input('random number seed -> ')
random.seed(seed)
```

```
11
12
13 def distance(p):
14     s = 0.0
15     (xprev, yprev) = p[-1] # previous town location
16     for i in range(N):
17         (x, y) = p[i] # current town location
18         s += ((x-xprev)**2+(y-yprev)**2)**0.5
19         (xprev, yprev) = (x, y)
20     return s
21
22
23 count = 0
24 def townswap(p):
25     # periodically draw the current state
26     global count
27     if count % 1000 == 0:
28         line.set_xdata([x for (x,y) in p+[p[0]]])
29         line.set_ydata([y for (x,y) in p+[p[0]]])
30         (T, _, d) = minimize.anneal.state
31         pylab.title('T=%g D=%g' % (T, d))
32         pylab.draw()
33     count += 1
34
35     # swap two distinct towns
36     (i, j) = (random.randrange(len(p)), random.randrange(len(p)))
37     while i == j: # must be distinct towns
38         (i, j) = (random.randrange(len(p)), random.randrange(len(p)))
39     p = list(p) # copy of p
40     (p[i], p[j]) = (p[j], p[i]) # swap towns i and j
41     return p
42
43
44 # plot initial path
45 pylab.ion()
46 pylab.figure(figsize=(5,5))
47 pylab.xlim(0, 1)
48 pylab.ylim(0, 1)
49 (line,) = pylab.plot([x for (x,y) in p+[p[0]]], [y for (x,y) in p+[p
       [0]]], 'o-')
50 pylab.draw()
51
52 print 'initial distance:', distance(p)
53
54 p, d = minimize.anneal(distance, p, townswap)
55
56 print 'final distance:', d
57
58 # plot final path
59 line.set_xdata([x for (x,y) in p+[p[0]]])
```

```
60 line.set_ydata([y for (x,y) in p+[p[0]]])
61 (T, _, _) = minimize.anneal.state
62 pylab.title('T=%g D=%g' % (T, d))
63 pylab.draw()
64 pylab.ioff()
65 pylab.show()
```

A.4 Interpolation

If we have a function $f(x)$ whose value we know at two points, x_0 and x_1, i.e., we know $y_0 = f(x_0)$ and $y_1 = f(x_1)$, then we can construct a line $y = y_0 + (x - x_0)(y_1 - y_0)/(x_1 - x_0)$ that passes through the points (x_0, y_0) and (x_1, y_1). This line is a linear approximation to the original function $f(x)$, and if we want to estimate the value of $f(x)$ for some value of x in the $x_0 < x < x_1$ we can use the value given by our linear approximation. This is known as *linear interpolation*.

If we have values of the function at more points then it will be possible to fit a higher order polynomial to these points and our polynomial approximation to the function will presumably be better. Suppose we have a function $f(x)$ that we have evaluated at a N points, $\{x_i\}$ for $0 \le i \le N-1$, and that the known values are $y_i = f(x_i)$. Then *Lagrange's interpolation formula* finds the polynomial of degree $N-1$ that passes through these points. If we want to evaluate the function for some x that is *not* one of the $\{x_i\}$, we have

$$f(x) = \sum_{i=0}^{N-1} \lambda_i(x) y_i + R_{N-1}(x) \tag{A.36}$$

where

$$\lambda_i(x) = \prod_{\substack{j=0 \\ j\neq i}}^{N-1} \frac{x - x_j}{x_i - x_j} = \frac{(x-x_0)\cdots(x-x_{i-1})(x-x_{i+1})\cdots(x-x_{N-1})}{(x_i-x_0)\cdots(x_i-x_{i-1})(x_i-x_{i+1})\cdots(x_i-x_{N-1})} \tag{A.37}$$

and $R_{N-1}(x)$ is a remainder term

$$|R_{N-1}(x)| \le \frac{(x_{N-1}-x_0)^N}{N!} \max_{x_0 \le x \le x_{N-1}} |f^{(N)}(x)|. \tag{A.38}$$

Here, $f^{(N)}(x)$ is the Nth derivative of f. Notice that $\lambda_i(x_j) = \delta_{ij}$ so the polynomial that is used in the approximation does in fact pass through all the known points.

The most basic case is the two-point interpolation formula, or linear interpolation. In this case, $n = 1$, and we use the two values $y_0 = f(x_0)$ and $y_1 = f(x_1)$ to interpolate the function at the value x:

$$\begin{aligned} f(x) &\approx \frac{x - x_1}{x_0 - x_1}\, y_0 + \frac{x - x_0}{x_1 - x_0}\, y_1 \\ &= y_0 + \frac{y_1 - y_0}{x_1 - x_0}\,(x - x_0) \end{aligned} \tag{A.39}$$

which is the same formula for the linear approximation that we had before.

An efficient way to performing polynomial interpolation is given by *Neville's algorithm*. Suppose

$$\overset{n}{y}_i(x)$$

are polynomials of degree n that pass through the points $y_i, y_{i+1}, \ldots, y_{i+n}$. When $n=0$ the polynomials are degree 0 (constants), and we have

$$\overset{0}{y}_i(x) = y_i \quad \text{(constant)}$$

for $i = 0, \ldots, N-1$. When $n=1$ the polynomials are degree 1 (lines), and we have

$$\overset{1}{y}_i(x) = \frac{x - x_{i+1}}{x_i - x_{i+1}} \overset{0}{y}_i + \frac{x - x_i}{x_{i+1} - x_i} \overset{0}{y}_{i+1}$$

for $i = 0, \ldots, N-2$. Then, a recurrence relation gives the $N-n$ degree n polynomials in terms of the degree $(n-1)$ polynomials as

$$\overset{n}{y}_i(x) = \frac{x - x_{i+n}}{x_i - x_{i+n}} \overset{n-1}{y}_i + \frac{x - x_i}{x_{i+n} - x_i} \overset{n-1}{y}_{i+1} \tag{A.40}$$

for $i = 0, \ldots, N-n-1$. The result we seek then is for the $n = N-1$ degree polynomial with $i=0$:

$$\overset{N-1}{y}_0(x).$$

An example interpolation routine, `polynomial`, that implements Neville's algorithm is given in the listing for the module `interpolate.py`.

Listing A.5: Module `interpolate.py`

```
1  def polynomial(xx, x, y):
2      """ Polynomial interpolation of points (x[i],y[i]) to find y(xx).
3      Warning: the values of y[] are modified."""
4
5      N = len(x)
6      for n in range(1, N):
7          for i in range(N-n):
8              y[i] = (xx-x[i])*y[i+1]+(x[i+n]-xx)*y[i]
9              y[i] /= x[i+n]-x[i]
10     return y[0]
```

A.5 Integration

We seek to compute the integral

$$Q = \int_a^b f(x)\,dx \tag{A.41}$$

by the method of *quadrature*. A direct approach is to use the *Riemann sum*

$$Q \approx S = \Delta x \sum_{n=0}^{N-1} f(a + n\,\Delta x) \tag{A.42}$$

with $\Delta x = (b-a)/N$. As $N \to \infty$ (so $\Delta x \to 0$) this approach will converge to the value of the integral. To find the accuracy of this *rectangular method*, consider a single step:

$$\begin{aligned}\int_{n\,\Delta x}^{(n+1)\Delta x} f(x)\,dx &= \int_{n\,\Delta x}^{(n+1)\Delta x} \Big[f(n\,\Delta x) + f'(n\,\Delta x)(x - n\,\Delta x) \\ &\qquad + \frac{1}{2} f''(n\,\Delta x)(x - n\,\Delta x)^2 + \cdots \Big] dx \\ &= f(n\,\Delta x)\,\Delta x + \frac{1}{2} f'(n\,\Delta x)(\Delta x)^2 + O(\Delta x^3) \\ &= f(n\,\Delta x)\,\Delta x + \frac{1}{2} f'(\xi_n)(\Delta x)^2\end{aligned} \tag{A.43}$$

where ξ_n is some number between $n\,\Delta x$ and $(n+1)\Delta x$. The first term is just the contribution to the Riemann sum while the second term represents extra part that is ignored in the Riemann sum — the error term. But such an error is introduced by *each* of the terms in the Riemann sum, so the accumulated error will be a factor of $\sim N = (b-a)/\Delta x$ larger. We therefore find

$$\int_a^b f(x)\,dx = \Delta x \sum_{n=0}^{N-1} f(a + n\,\Delta x) + O\left(\frac{(b-a)^2 f'(\xi)}{N} \right). \tag{A.44}$$

Doubling the number of points used in the sum (i.e., the number of times that the function is evaluated) merely doubles the accuracy of the result.

An obvious improvement is to evaluate the function at the midpoint between two steps:

$$Q \approx S = \Delta x \sum_{n=0}^{N-1} f\left(a + (n + \tfrac{1}{2})\Delta x\right). \tag{A.45}$$

This is known as the *midpoint method*, and it is convenient for computing open integrals where we do not wish to evaluate the function at the endpoints a or b (e.g., if the integrand is singular at these points). The accuracy of each step of the midpoint method can be found:

$$\begin{aligned}\int_{n\,\Delta x}^{(n+1)\Delta x} f(x)\,dx &= \int_{n\,\Delta x}^{(n+1)\Delta x} f\left((n+\tfrac{1}{2})\Delta x\right) dx \\ &\qquad + \int_{n\,\Delta x}^{(n+1)\Delta x} f'\left((n+\tfrac{1}{2})\Delta x\right)\left(x - (n+\tfrac{1}{2})\Delta x\right) dx \\ &\qquad + O(\Delta x^3) \\ &= f\left((n+\tfrac{1}{2})\Delta x\right)\Delta x + O(\Delta x^3)\end{aligned} \tag{A.46}$$

where we note that the second integral on the right-hand-side of vanishes. Thus, for the extended midpoint method,

$$\int_a^b f(x)\,dx = \Delta x \sum_{n=0}^{N-1} f\left(a + (n+\tfrac{1}{2})\Delta x\right) + O\left(\frac{(b-a)^3 f''(\xi)}{N^2}\right). \tag{A.47}$$

We see that doubling of the number of points increases the accuracy by a factor of four.

For closed integrals, where the endpoint is to be retained, the *trapezoid method* approximates the integral at each step as a trapezoid rather than a rectangle:

$$\int_{n\,\Delta x}^{(n+1)\Delta x} f(x)\,dx \approx \Delta x \left[\frac{1}{2} f\left(n\,\Delta x\right) + \frac{1}{2} f\left((n+1)\Delta x\right)\right]. \tag{A.48}$$

It is easy to show by Taylor expanding the function about both $n\,\Delta x$ and $(n+1)\Delta x$ that the accuracy of this step is $O(\Delta x^3)$ so the extended trapezoid method is

$$\begin{aligned}\int_a^b f(x)\,dx = \Delta x\Big[\frac{1}{2}f(a) + f(a+\Delta x) + f(a+2\Delta x) + \cdots \\ + f(a+(N-1)\Delta x) + \frac{1}{2}f(a+N\Delta x)\Big] \\ + O\left(\frac{(b-a)^3 f''(\xi)}{N^2}\right).\end{aligned} \tag{A.49}$$

Again the method is $O(1/N^2)$. Figure A.2 illustrates the rectangle, midpoint, and trapezoid quadrature methods.

A useful property of the trapezoid rule is that if you have used it to obtain a value S with $N+1$ points, then if you wish to (roughly) double the number of points to $2N+1$, you already have $N+1$ of those points computed. Therefore, the extended trapezoid method can be written as this recurrence: on stage $n=0$ with $(\Delta x)_0 = b-a$ the value

$$S_0 = (b-a)\,\frac{f(a)+f(b)}{2} \tag{A.50}$$

is computed; then at each later stage we have

$$S_{n+1} = \frac{1}{2}\left[S_n + (\Delta x)_{n+1} \sum_{i=0}^{2^n-1} f\left(a + \tfrac{1}{2}(\Delta x)_{n+1} + i\,(\Delta x)_{n+1}\right)\right] \tag{A.51}$$

where $(\Delta x)_{n+1} = (b-a)/2^n$. This recurrence is implemented in the routine `trapstep` in `integrate.py` (see Listing A.6). You can use it directly to compute an integral with $n_{\max}$ levels of refinement as

```
s = 0.0
for n in range(nmax):
        s = integrate.trapstep(f, a, b, n, s)
```

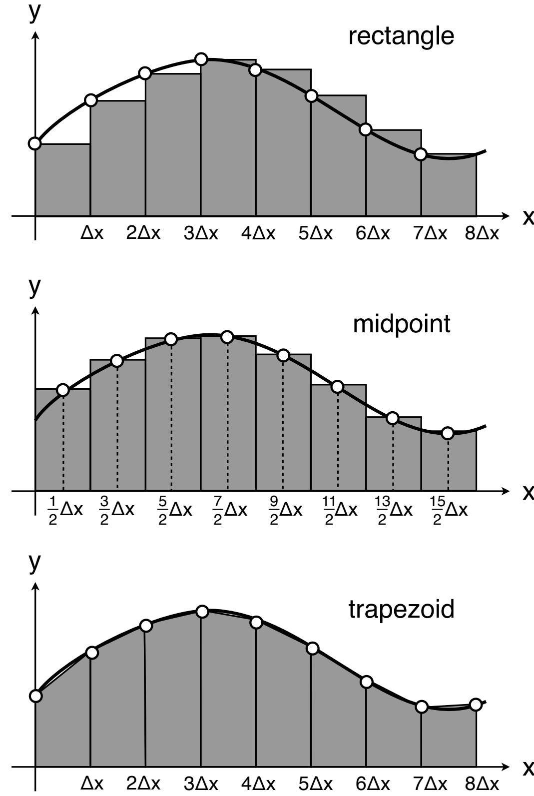

Figure A.2: The rectangle, midpoint, and trapezoid methods of quadrature. The open circles indicate the points where the function is evaluated.

The routine `trapezoid` in Listing A.6 performs this but continues the refinement until the desired accuracy is achieved.

An important feature about the extended trapezoid rule derives from its connection to the *Euler-Maclaurin formula*,

$$
\begin{aligned}
\Delta x &\left[\frac{1}{2}f(a)+f(a+\Delta x)+\cdots+f\,(a+(N-1)\Delta x)+\frac{1}{2}f(b)\right] \\
&= \int_a^b f(x)\,dx + \sum_{k=1}^{p}\frac{B_{2k}}{(2k)!}\,(\Delta x)^{2k}\left(f^{(2k-1)}(b)-f^{(2k-1)}(a)\right)+R_p
\end{aligned}
\tag{A.52}
$$

where B_{2k} are the Bernoulli numbers and R_p is a remainder term that is on the order of the next term in the summation on the right hand side. The left hand side of this equation is the extended trapezoid rule and the sum on the right hand side represents the error. As expected, the first term in the sum is $O(\Delta x^2)$. However,

notice that the summation includes only even powers of Δx, and, in particular, there is no Δx^3 term. This means that if we compute the nth stage of refinement of the extended trapezoid rule, S_n, then

$$S_n = \int_a^b f(x)\,dx + \frac{B_2}{2!}(\Delta x)_n^2\big(f'(b) - f'(a)\big) + O((\Delta x)_n^4) \tag{A.53a}$$

and for the $(n+1)$th stage

$$S_{n+1} = \int_a^b f(x)\,dx + \frac{B_2}{2!}(\Delta x)_{n+1}^2\big(f'(b) - f'(a)\big) + O((\Delta x)_{n+1}^4). \tag{A.53b}$$

But since $(\Delta x)_n = 2(\Delta x)_{n+1}$, we find

$$4S_{n+1} - S_n = 3\int_a^b f(x)\,dx + O((\Delta x)_n^4) \tag{A.54}$$

or

$$\int_a^b f(x)\,dx = \frac{1}{3}(4S_{n+1} - S_n) + O((\Delta x)_n^4). \tag{A.55}$$

Thus we have constructed an $O(1/N^4)$ method! This is known as the *extended Simpson's rule*. Explicitly, it is

$$\begin{aligned}\int_a^b f(x)\,dx = \Delta x\Big[&\frac{1}{3}f(a) + \frac{4}{3}f(a+\Delta x) + \frac{2}{3}f(a+2\Delta x) + \frac{4}{3}f(a+3\Delta x) + \cdots \\ &+ \frac{2}{3}f\,(a+(N-2)\Delta x) + \frac{4}{3}f\,(a+(N-1)\Delta x) + \frac{1}{3}f(b)\Big] \\ &+ O\left(\frac{(b-a)^5 f^{(4)}(\xi)}{N^4}\right).\end{aligned} \tag{A.56}$$

The routine `simpson` in Listing A.6 employs Eq. (A.55) to perform a quadrature with continued refinement until the desired accuracy is achieved. Since the error is $O(1/N^4)$ in routine `simpson`, it converges much faster than routine `trapezoid`, where the error is $O(1/N^2)$.

There is nothing to stop us from using the same trick to generate higher and higher order methods for integration. For example, if we have the result from stage $n-1$, S_{n-1}, with $(\Delta x)_{n-1} = 2(\Delta x)_n = 4(\Delta x)_{n+1}$, we can use it to eliminate the $O((\Delta x)_n^4)$ term in Eq. (A.54). We have:

$$S_{n+1} = Q + A_1(\Delta x)_{n+1}^2 + A_2(\Delta x)_{n+1}^4 + O(\Delta x^6) \tag{A.57a}$$

$$S_n = Q + 4A_1(\Delta x)_{n+1}^2 + 16A_2(\Delta x)_{n+1}^4 + O(\Delta x^6) \tag{A.57b}$$

$$S_{n-1} = Q + 16A_1(\Delta x)_{n+1}^2 + 256A_2(\Delta x)_{n+1}^4 + O(\Delta x^6) \tag{A.57c}$$

where Q is the integral we are computing and A_1 and A_2 are the coefficients of the $O(\Delta x^2)$ and $O(\Delta x^4)$ error terms respectively. We find

$$45Q = 64S_{n+1} - 20S_n + S_{n-1} + O\left((\Delta x)_n^6\right) \tag{A.58}$$

or

$$\int_a^b f(x)\,dx = \frac{1}{45}\left[64S_{n+1} - 20S_n + S_{n-1}\right] + O\left(\frac{(b-a)^7 f^{(6)}(\xi)}{N^6}\right) \tag{A.59}$$

where $N = 2^n$. This is known as Boole's rule. Explicitly it is

$$\begin{aligned}\int_a^b f(x)\,dx = \Delta x\Big[&\frac{14}{45}f(a) + \frac{64}{45}f(a+\Delta x) + \frac{24}{45}f(a+2\Delta x) + \frac{64}{45}f(a+3\Delta x)\\ &+ \frac{28}{45}f(a+4\Delta x) + \frac{64}{45}f(a+5\Delta x) + \frac{24}{45}f(a+6\Delta x) + \cdots\\ &+ \frac{24}{45}f\left(a+(N-2)\Delta x\right) + \frac{64}{45}f\left(a+(N-1)\Delta x\right) + \frac{14}{45}f(b)\Big]\\ &+ O\left(\frac{(b-a)^7 f^{(6)}(\xi)}{N^6}\right).\end{aligned} \tag{A.60}$$

We could continue this procedure, but notice what is happening: we have a sequence of refinements of the trapezoid rule and with each extra term in the sequence we can eliminate one more error term. The remaining error term is then $O(1/N^{2(n+1)})$. We express the Euler-Maclaurin formula as a polynomial of degree n,

$$P(\xi) = A_0 + A_1\,\xi + A_2\,\xi^2 + \cdots + A_n\,\xi^n, \tag{A.61}$$

where $A_0 = Q + R_n$ is the value of the integral we seek, Q, plus a remainder term, R_n, $A_1, A_2, \ldots, A_n$ are some constant coefficients, and $\xi = \Delta x^2$. A set of refinements, $\{S_0, S_1, \ldots, S_n\}$, are simply different samples of this polynomial with $S_n = P(\xi_n)$, and $\xi_n = 2^{-2n}(b-a)^2$, so we use polynomial extrapolation to $\xi = 0$ (i.e., $\Delta x = 0$) to obtain the value $A_0 = Q + R_n$ where the remaining error, R_n, is $O(\Delta x^{2(n+1)})$. This technique is known as *Richardson extrapolation*. For example, suppose we have $\{S_0, S_1, S_2\}$ which are the values of the polynomial $P(\xi)$ at the points $\{\xi_0, \xi_1 = \frac{1}{4}\xi_0, \xi_2 = \frac{1}{16}\xi_0\}$ and where $\xi_0 = (b-a)^2$. Then the interpolating polynomial is

$$P(\xi) = \frac{(\xi - \frac{1}{4})(\xi - \frac{1}{16})}{(1 - \frac{1}{4})(1 - \frac{1}{16})}\,S_0 + \frac{(\xi - 1)(\xi - \frac{1}{16})}{(\frac{1}{4} - 1)(\frac{1}{4} - \frac{1}{16})}\,S_1 + \frac{(\xi - 1)(\xi - \frac{1}{4})}{(\frac{1}{16} - 1)(\frac{1}{16} - \frac{1}{4})}\,S_2 \tag{A.62}$$

and the extrapolation to $\xi = 0$ yields

$$Q = P(0) - R_2 = \frac{1}{45}\,S_0 - \frac{20}{45}\,S_1 + \frac{64}{45}\,S_2 - R_2 \tag{A.63}$$

where $R_2 \sim O(\Delta x^6)$ [cf. Eq. (A.59)]. In this way, as we increase the number of stages of refinement, n, we also increase the degree of the polynomial and remove successively more error terms with our extrapolation. The technique of using Richardson extrapolation to accelerate convergence of the trapezoidal rule for quadrature is known as *Romberg integration*. The routine `romberg` in Listing A.6 uses the Romberg method to evaluate the integral.

The quadrature rules thus far have been of a form

$$\int_a^b f(x)\,dx \approx \sum_{n=0}^{N-1} w_n f(x_n) \tag{A.64}$$

where w_n are a set of weight coefficients and x_n have been evenly spaced points between a and b. If we relax the restriction that the spacing of the points be even, then we can make optimal choices of both w_n and x_n, which allows us to obtain higher-order quadrature rules without increasing the number of function evaluations. The method is known as *Gaussian quadratures*.

To illustrate the general idea, suppose we wish to compute the integral

$$\int_{-1}^{1} f(x)\,dx$$

where $f(x)$ is a cubic polynomial

$$f(x) = c_0 + c_1 x + c_2 x^2 + c_3 x^3$$

for some set of coefficients $\{c_0, c_1, c_2, c_3\}$. Of course, the result is going to be

$$\int_{-1}^{1} f(x)\,dx = 2c_0 + \frac{2}{3}c_2$$

so it will be sufficient to determine this combination of the coefficients in order to evaluate the integral. These can be determined by evaluating the function at judiciously selected points:

$$f(-\sqrt{1/3}) + f(\sqrt{1/3}) = 2c_0 + \frac{2}{3}c_2$$

so

$$\int_{-1}^{1} f(x)\,dx = f(-\sqrt{1/3}) + f(\sqrt{1/3}).$$

Thus, the integral of any cubic polynomial can be exactly computed by evaluating it at two points. We will see that the integral of a polynomial of degree $2N-1$ can computed by evaluating it at N specially chosen points. If a function can be approximated as a polynomial, this provides a efficient way of evaluating its integral.

Consider an interpolating polynomial as described in Sec. A.4:

$$p(x) = \sum_{n=0}^{N-1} \lambda_n(x) f(x_n) \tag{A.65}$$

where $\lambda_n(x)$ are Lagrange's interpolating polynomials given in Eq. (A.37). If $f(x)$ is a polynomial of degree $N-1$ then $p(x)$ is exactly equal to $f(x)$; otherwise, $p(x)$ is a polynomial approximation to $f(x)$. If we replace $f(x)$ in the integrand by $p(x)$ we find

$$\int_a^b f(x)\,dx \approx \int_a^b p(x)\,dx = \sum_{n=0}^{N-1} w_n f(x_n) \tag{A.66}$$

with

$$w_n = \int_a^b \lambda_n(x)\,dx. \tag{A.67}$$

Once we choose the evaluation points $\{x_n\}$ for $0 \le n \le N-1$ we can evaluate these weight coefficients. The evaluation points are known as *abscissas*.

Suppose that $f(x)$ is a polynomial of degree $2N-1$. Then it can be factored as

$$f(x) = q(x)P_N(x) + r(x) \tag{A.68}$$

where $P_N(x)$ is a polynomial of degree N, $q(x)$ is a quotient polynomial of degree $N-1$, and $r(x)$ is a remainder polynomial also of degree $N-1$. We choose $P_N(x)$ to be a polynomial that is orthogonal to all polynomials of degree less than N in the interval $a \le x \le b$:

$$\int_a^b x^n P_N(x)\,dx = 0 \quad \text{for } n = 0, 1, \ldots, N-1. \tag{A.69}$$

Then we have

$$\int_a^b f(x)\,dx = \int_a^b r(x)\,dx = \sum_{n=0}^{N-1} w_n r(x_n) \tag{A.70}$$

and notice that the second equality is exact because $r(x)$ is a polynomial of degree $N-1$ or less. Also we have

$$\sum_{n=0}^{N-1} w_n f(x_n) = \sum_{n=0}^{N-1} w_n q(x_n) P_N(x_n) + \sum_{n=0}^{N-1} w_n r(x_n). \tag{A.71}$$

Because $P_N(x)$ is of degree N, it has N roots. If we choose the $\{x_n\}$ for $n = 0, 1, \ldots, N-1$ to be those roots, then the first term on the right hand side vanishes and we have

$$\sum_{n=0}^{N-1} w_n f(x_n) = \sum_{n=0}^{N-1} w_n r(x_n). \tag{A.72}$$

Combining this with Eq. (A.70), we have

$$\int_a^b f(x)\,dx = \sum_{n=0}^{N-1} w_n f(x_n) \tag{A.73}$$

as an exact equality when $f(x)$ is a polynomial of degree $2N-1$.

The method therefore requires us to find a polynomial of degree N, $P_N(x)$, that is (i) orthogonal to all lower-degree polynomials in the interval $a \le x \le b$, and (ii) has all N roots in this interval. These roots are then used as the locations at which the function $f(x)$ is to be evaluated, and as the points at which to construct the Lagrange interpolating polynomial $\lambda_n(x)$. Notice that

$$P_N(x) = A_N(x-x_0)(x-x_1)\cdots(x-x_{N-1}) \tag{A.74}$$

where A_N is some constant, so, taking one derivative and evaluating at the root x_n, we have

$$P'_N(x_n) = A_N(x_n-x_0)\cdots(x_n-x_{n-1})(x_n-x_{n+1})\cdots(x_n-x_{N-1}). \tag{A.75}$$

Now we see that the Lagrange interpolating polynomial of Eq. (A.37) can be expressed as

$$\lambda_n(x) = \frac{P_N(x)}{(x-x_n)P'_N(x_n)} \tag{A.76}$$

and therefore the weights can be written as

$$w_n = \frac{1}{P'_N(x_n)} \int_a^b \frac{P_N(x)}{x-x_n}\,dx. \tag{A.77}$$

This formula for the weights can be put into a more convenient form. First, note that, for some integer k with $0 \le k \le N-1$,

$$x_n^k \int_a^b \frac{P_N(x)}{x-x_n}\,dx = \int_a^b \left[\frac{x^k}{x-x_n} - \frac{x^k - x_n^k}{x-x_n}\right] P_N(x)\,dx = \int_a^b x^k \frac{P_N(x)}{x-x_n}\,dx \tag{A.78}$$

where the second equality holds because the second term in square brackets is a polynomial of degree $k < N$, and is orthogonal to $P_N(x)$. Any linear combination of terms of this equation can be constructed, and, in particular

$$P_{N-1}(x_n) \int_a^b \frac{P_N(x)}{x-x_n}\,dx = \int_a^b P_{N-1}(x) \frac{P_N(x)}{x-x_n}\,dx. \tag{A.79}$$

Now $P_N(x)/(x-x_n)$ is a polynomial of degree $N-1$, which can be written as

$$\frac{P_N(x)}{x-x_n} = \frac{A_N}{A_{N-1}} P_{N-1}(x) + s(x) \tag{A.80}$$

where $s(x)$ is a polynomial of degree $N-2$, which is orthogonal to $P_{N-1}(x)$. Hence, Eq. (A.79) can be rewritten

$$\int_a^b \frac{P_N(x)}{x-x_n}\,dx = \frac{A_N/A_{N-1}}{P_{N-1}(x_n)}\int_a^b [P_{N-1}(x)]^2\,dx. \tag{A.81}$$

Using this identitity in Eq. (A.77) we obtain

$$w_n = \frac{A_N/A_{N-1}}{P_N'(x_n)P_{N-1}(x_n)}\int_a^b [P_{N-1}(x)]^2\,dx. \tag{A.82}$$

The set of orthogonal polynomials we are interested in are the *Legendre polynomials*. These are the polynomial solutions to *Legendre's differential equation*

$$\frac{d}{dx}\left[(1-x^2)\frac{d}{dx}P_N(x)\right] + N(N+1)P_N(x) = 0 \tag{A.83}$$

and can be obtained using *Rodrigues' formula*

$$P_N(x) = \frac{1}{2^N N!}\frac{d^N}{dx^N}[(x^2-1)^N]. \tag{A.84}$$

The first few Legendre polynomials along with their abscissas and weights are tabulated in Table A.1. These polynomials are orthogonal in the interval $-1 \le x \le 1$ with

$$\int_{-1}^{1} P_M(x)P_N(x) = \frac{2}{2N+1}\delta_{MN}. \tag{A.85}$$

Although our interval was $a \le x \le b$, these polynomials are suitable since we can make the change of variables

$$\int_a^b f(x)\,dx = \frac{b-a}{2}\int_{-1}^{1} f(\tfrac{1}{2}(b-a)x' + \tfrac{1}{2}(b+a))\,dx'. \tag{A.86}$$

Therefore, without loss of generality, we take $a=-1$ and $b=1$ going forward.

Our task is simply to compute the absiccas and weights for the Legendre polynomials. The abscissas are the roots of the Legendre polynomials. We can compute these numerically using Newton's method described in Sec. A.2. We need approximate locations for the roots to use as initial guesses. From Eq. (22.16.6) of Abramowitz and Stegun (1964) we take the initial guesses for the roots to be

$$x_n \approx \cos\left(\frac{4n+3}{4N+2}\pi\right) \tag{A.88}$$

When performing Newton's method, we need to evaluate $P_N(x)$ and $P_N'(x)$. These can be obtained using the recurrence relationships

$$P_{N+1}(x) = \frac{(2N+1)xP_N(x) - N\,P_{N-1}(x)}{N+1} \tag{A.89}$$

N	Abscissas	Weights	Legendre polynomial	
0	–	–	$P_0(x) = 1$	(A.87a)
1	0	2	$P_1(x) = x$	(A.87b)
2	$-\sqrt{\frac{1}{3}}, +\sqrt{\frac{1}{3}}$	1, 1	$P_2(x) = \frac{1}{2}(3x^2 - 1)$	(A.87c)
3	$-\sqrt{\frac{3}{5}}, 0, +\sqrt{\frac{3}{5}}$	$\frac{5}{9}, \frac{8}{9}, \frac{5}{9}$	$P_3(x) = \frac{1}{2}(5x^3 - 3x)$	(A.87d)

Table A.1: Legendre polynomials and their abscissas and weights.

and

$$P'_N(x) = N \frac{xP_N(x) - P_{N-1}(x)}{x^2 - 1} \tag{A.90}$$

beginning with the first two Legendre polynomials given by Eqs. (A.87a) and (A.87b). Note that, at a root of $P_N(x)$ we have

$$P'_N(x_n) = N \frac{P_{N-1}(x_n)}{1 - x_n^2}. \tag{A.91}$$

Thus, the Newton's method will update our guesses with

$$\overset{i+1}{x}_n = \overset{i}{x}_n - \frac{1 - \overset{i}{x}{}_n^2}{N} \frac{P_N(\overset{i}{x}_n)}{P_{N-1}(\overset{i}{x}_n)}. \tag{A.92}$$

To compute the weights, we can use Eq. (A.85) and Eq. (A.91). In addition, from Rodrigues' formula, Eq. (A.84), we see

$$A_N = \frac{(2N)!}{2^N (N!)^2}. \tag{A.93}$$

Combining these results with Eq. (A.82) we have finally

$$w_n = \frac{2}{N^2} \frac{1 - x_n^2}{[P_{N-1}(x_n)]^2}. \tag{A.94}$$

Multidimensional integration can be performed by successive applications of these one-dimensional integrations. However, for high-dimensional integrals, the computational cost can become prohibitive. For example, if a one dimensional integral requires N function calls, an M-dimensional integral requires on order of N^M function calls, which can be huge if M is large. An alternative to evaluating the function on a fixed grid of points is to evaluate the function at *random* points in

the domain of integration. This is known as *Monte Carlo integration*. Then, the multidimensional integral can be evaluated as

$$\int_\Omega f(\boldsymbol{x})d^M\boldsymbol{x} \simeq Q_N \tag{A.95}$$

for large N where

$$Q_N = \operatorname{Vol}\Omega\,\frac{1}{N}\sum_{n=0}^{N-1} f(\boldsymbol{x}_n) \tag{A.96}$$

and where $\{\boldsymbol{x}_n\}$ for $0 \le n \le N-1$ are a set of random points within the M-dimensional domain Ω whose volume is $\operatorname{Vol}\Omega$. An estimate of the error in the estimate is given by

$$\operatorname{Var}(Q_N) = \langle Q_N^2\rangle - \langle Q_N\rangle^2 \approx (\operatorname{Vol}\Omega)^2\,\frac{s_f^2}{N} \tag{A.97}$$

where

$$s_f^2 = \frac{1}{N-1}\sum_{n=0}^{N-1}\left(f(\boldsymbol{x}_n) - \frac{1}{N}\sum_{n'=0}^{N-1} f(\boldsymbol{x}_{n'})\right)^2 \tag{A.98}$$

is the square of the sample standard deviation of the function values. Thus, for large N, the error in the Monte Carlo estimate of the integral becomes

$$E_N \approx \sqrt{\operatorname{Var}(Q_N)} = \operatorname{Vol}\Omega\,\frac{s_f}{\sqrt{N}}. \tag{A.99}$$

We see that convergence is relatively slow as the error decreases only as the square-root of the number of function evaluations. However, this is true regardless of the dimensions of the integral.

The listing for `integrate.py` contains routines to perform integration of a function using the extended trapezoid rule, `trapezoid`, the extended Simpson's rule, `simpson`, and the Romberg method, `romberg`. These are all based on successive refinements of the trapezoid rule which is provided in the routine `trapstep`. The routine `abscissas` computes the abscissas and weights needed for Gaussian quadrature, and the routine `gaussian` performs integration using Gaussian quadratures. In addition, two routines for performing multidimensional integration are provided: `gaussiannd` and `montecarlo`.

Listing A.6: Module `integrate.py`

```
1  import interpolate, pylab, itertools
2
3
4  def trapstep(f, a, b, n, s):
5      """ Perform the nth refinement of a trapezoid method and update sum.
6      Here, f is the integrand, a and b are the limits of the integral,
7      n is the step (0 on first call), and s is the sum so far. """
8
```

```
9      if n == 0:
10         return (b−a)∗(f(b)+f(a))/2.0
11     else:
12         steps = 2∗∗(n−1)
13         dx = (b−a)/steps
14         x0 = a+dx/2.0
15         s = (s+dx∗sum(f(x0+i∗dx) for i in range(steps)))/2.0
16         return s
17 
18 
19 def trapezoid(f, a, b, eps=1e−6):
20     """ Trapezoidal method of quadrature to a specified accuracy.
21     Here, f is the integrand, a and b are the limits of the integral,
22     and eps is the desired fractional accuracy. """
23 
24     n = 0
25     s = trapstep(f, a, b, n, 0.0)
26     while True:
27         n += 1
28         s0 = s  # s0 is the previous value of the trapezoid sum
29         s = trapstep(f, a, b, n, s)
30         if abs(s−s0) < eps∗abs(s0):
31             return s
32 
33 
34 def simpson(f, a, b, eps=1e−6):
35     """ Simpson’s method of quadrature to a specified accuracy.
36     Here, f is the integrand, a and b are the limits of the integral,
37     and eps is the desired fractional accuracy. """
38 
39     n = 0
40     q = s = trapstep(f, a, b, n, 0.0)
41     while True:
42         n += 1
43         (q0, s0) = (q, s)  # save previous values
44         s = trapstep(f, a, b, n, s)
45         q = (4.0∗s−s0)/3.0
46         if abs(q−q0) < 0.25∗eps∗abs(q0):
47             return q
48 
49 
50 def romberg(f, a, b, eps=1e−6):
51     """ Romberg’s method of quadrature to a specified accuracy.
52     Here, f is the integrand, a and b are the limits of the integral,
53     and eps is the desired fractional accuracy. """
54 
55     degree = 5  # degree of polynomial extrapolation
56     n = 0
57     s = [trapstep(f, a, b, n, 0.0)]
58     xi = [1.0]
```

```
59      while True:
60          n += 1
61          s += [trapstep(f, a, b, n, s[-1])]  # append new element
62          xi += [xi[-1]/4.0]  # append new element
63          if len(s) >= degree:
64              ss = s[-degree:]  # last degree elements of s
65              # extrapolate to dx = 0
66              q = interpolate.polynomial(0.0, xi[-degree:], ss)
67              dq = ss[1]-ss[0]  # error in extrapolation
68              if abs(dq) < eps*abs(q):
69                  return q
70
71
72  def abscissas(N):
73      """ Returns a list of (abscissa, weight) pairs suitable for
74      Gaussian quadrature. """
75
76      if N == 0:
77          return []
78      if N == 1:
79          return [(0.0, 2.0)]
80      # initial guess of root locations
81      x = pylab.cos(pylab.pi*(4.0*pylab.arange(N)+3.0)/(4.0*N+2.0))
82      # Newton's method
83      eps = 1e-15
84      while True:
85          # generate Legendre polynomials using recurrence relation
86          (p0, p1) = (1.0, x)
87          for n in range(1,N):
88              (p0, p1) = (p1, ((2.0*n+1.0)*x*p1-n*p0)/(n+1.0))
89          dx = -(1.0-x**2)*p1/(N*p0)
90          x += dx
91          if max(abs(dx)) < eps:
92              break
93      w = 2.0*(1.0-x**2)/(N*p0)**2
94      return [(x[n], w[n]) for n in range(N)]
95
96
97  def gaussian(f, a, b, N=5):
98      """ Gaussian quadrature with a specified number of abscissas.
99      Here, f is the integrand, a and b are the limits of the integral,
100     and N is the number of abscissas. """
101
102     fac = 0.5*(b-a)
103     mid = 0.5*(b+a)
104     return fac*sum(w*f(fac*x+mid) for (x,w) in abscissas(N))
105
106
107 def gaussiannd(f, a, b, N=5):
108     """ Multidimensional Gaussian quadrature.
```

```
109     Here, f is the integrand, a and b are arrays giving the limits
110     of the integral, and N is the number of abscissas. """
111
112     a = pylab.asarray(a)
113     b = pylab.asarray(b)
114     ndim = a.size
115     if a.size == 1:  # use normal 1d Gaussian quadrature
116         return gaussian(f, a, b)
117     fac = 0.5*(b-a)
118     mid = 0.5*(b+a)
119     s = 0.0
120     # loop over all possible ndim-vectors of abscissas
121     for xw in itertools.product(abscissas(N), repeat=ndim):
122         x = pylab.array([x for (x,_) in xw])
123         w = pylab.prod([w for (_,w) in xw])
124         s += w*f(fac*x+mid)
125     return pylab.prod(fac)*s
126
127
128 def montecarlo(f, a, b, eps=1e-3, nmin=100, nmax=1000000):
129     """ Monte Carlo integration.
130     Here, f is the integrand, a and b are arrays giving the limits
131     of the integral, and eps is the desired accuracy.
132     The parameters nmin and nmax specify the minimum and
133     maximum number of random points to use. """
134
135     a = pylab.asarray(a)
136     b = pylab.asarray(b)
137     vol = pylab.prod(b-a)
138     s = 0.0  # running average of f(x)
139     ssq = 0.0  # running sum of (f(x)-s)2
140     n = 0
141     while n < nmax:
142         n += 1
143         x = pylab.uniform(a, b)
144         fx = f(x)
145         d = fx - s
146         s += d/n
147         ssq += d*(fx - s)
148         err = ssq**0.5/n  # assume n-1 ~= n
149         if n > nmin and err < eps*abs(s):
150             break
151     return vol*s
```

As an example, consider the test integral

$$\int_0^{\pi} \sin(x)\, dx.$$

We can perform quadrature with the routines `trapezoid`, `simpson`, `romberg`,

`gaussian`, and `montecarlo` and count the number of function calls required using the following code

```
import pylab, integrate
pylab.seed(101)
def inttest(integrator):
    global count; count = 0
    def testfunction(x): global count; count += 1; return pylab.sin(x)
    print "result %.7f" % integrator(testfunction, 0.0, pylab.pi),
    print "obtained in %d function calls" % count,
    print "using %s" % integrator.__name__
inttest(integrate.trapezoid)
inttest(integrate.simpson)
inttest(integrate.romberg)
inttest(integrate.gaussian)
inttest(integrate.montecarlo)
```

and the resulting output is

```
result 1.9999996 obtained in 2049 function calls using trapezoid
result 2.0000000 obtained in 129 function calls using simpson
result 2.0000000 obtained in 17 function calls using romberg
result 2.0000001 obtained in 5 function calls using gaussian
result 2.0016163 obtained in 232504 function calls using montecarlo
```

Note that the accuracy requirement for the `montecarlo` routine is lower than for the other routines.

As an example of multidimensional integration, consider the 9-dimensional integral

$$\int_0^1 \cdots \int_0^1 f(x_0,\ldots,x_8)\,dx_0\cdots dx_8$$

with

$$f(x_0,\ldots,x_8) = \frac{1}{\sum_{n=0}^{8} x_n}.$$

We test the routines `gaussiannd` and `montecarlo` and measure the number of function calls with the following code

```
import pylab, integrate
pylab.seed(101)
def inttest9d(integrator):
    global count; count = 0
    def testfunction(x): global count; count += 1; return 1.0/sum(x)
    print "result %.7f" % integrator(testfunction, [0.0]*9, [1.0]*9),
    print "obtained in %d function calls" % count,
    print "using %s" % integrator.__name__
inttest9d(integrate.gaussiannd)
inttest9d(integrate.montecarlo)
```

if $x(t)$ is real, $\Im x(t)=0$,	then $\tilde{x}(-f)=\tilde{x}^*(f)$
if $x(t)$ is imaginary, $\Re x(t)=0$,	then $\tilde{x}(-f)=-\tilde{x}^*(f)$
if $x(t)$ is even, $x(t)=x(-t)$,	then $\tilde{x}(-f)=\tilde{x}(f)$
if $x(t)$ is odd, $x(t)=-x(-t)$,	then $\tilde{x}(-f)=-\tilde{x}(f)$

Table A.2: Symmetries of the Fourier transform.

and the resulting output is

```
result 0.2315219 obtained in 1953125 function calls using gaussiannd
result 0.2316330 obtained in 46677 function calls using montecarlo
```

Gaussian quadratures now requires many more function evaluations than the Monte Carlo method (note that $1953125 = 5^9$). As the dimensionality increases, use of `gaussiannd` quickly becomes impractical.

A.6 Fourier transform

The Fourier transform of a continuous function $x(t)$ is

$$\tilde{x}(f)=\int_{-\infty}^{\infty} \mathrm{e}^{-2\pi i f t} x(t)\,dt \tag{A.100}$$

while the inverse Fourier transform is

$$x(t)=\int_{-\infty}^{\infty} \mathrm{e}^{+2\pi i f t} \tilde{x}(f)\,df. \tag{A.101}$$

We say that $x(t)$ and $\tilde{x}(t)$ are Fourier transform pairs, and denote this relationship as

$$x(t) \iff \tilde{x}(f).$$

A number of important properties follow from the definition of the Fourier transform, and these are summarized in tables A.2 and A.3. In the time-domain, the *convolution* of two functions is

$$x * y=\int_{-\infty}^{\infty} x(\tau)y(t-\tau)\,d\tau \tag{A.102}$$

while the *correlation* of two functions is

$$\operatorname{corr}(x,y)=\int_{-\infty}^{\infty} x(t+\tau)y(\tau)\,d\tau. \tag{A.103}$$

scaling property:	$x(at) \Longleftrightarrow \frac{1}{\lvert a\rvert}\tilde{x}(f/a)$
shifting property:	$x(t-t_0) \Longleftrightarrow \tilde{x}(f)e^{-2\pi i f t_0}$
convolution theorem:	$(x * y)(t) \Longleftrightarrow \tilde{x}(f)\tilde{y}(f)$
correlation theorem:	$\mathrm{corr}(x,y)(t) \Longleftrightarrow \tilde{x}(f)\tilde{y}^*(f)$

Table A.3: Properties of the Fourier transform.

The total power in a signal is given in either the time- or the frequency-domains by integrating the modulus squared of the signal over the entire domain:

$$\int_{-\infty}^{\infty} |x(t)|^2 dt = \int_{-\infty}^{\infty} |\tilde{x}(f)|^2 df. \tag{A.104}$$

This is *Parseval's theorem*. We are often interested in the power spectral density, that is, the power that is contained in some frequency interval between f and $f + df$. A one-sided power spectral density folds the negative frequencies onto the positive frequencies so that only positive frequencies are required; it is defined as

$$S_x(f) = |\tilde{x}(f)|^2 + |\tilde{x}(-f)|^2. \tag{A.105}$$

The integral of $S_x(f)$ over all positive frequencies is then the total power in the signal.

If we sample the function at intervals Δt, so we have the samples $x_j = x(j\,\Delta t)$ of the original function, then we can recover the original function from the samples via

$$x(t) = \Delta t \sum_{j=-\infty}^{\infty} \frac{\sin[2\pi f_{\mathrm{Ny}}(t - j\,\Delta t)]}{\pi(t - j\,\Delta t)}\, x_j \tag{A.106}$$

where

$$f_{\mathrm{Ny}} = \frac{1}{2\Delta t} \tag{A.107}$$

is known as the *Nyquist frequency*. This result is known as the *sampling theorem*. However, the sampling theorem only holds if the function that was sampled, $x(t)$, was band limited to a frequency band that in which the highest frequency was less than the Nyquist frequency of the sampling. Otherwise the sampling process will result in *aliasing* high frequency content to low frequencies. Thus it is important to ensure that the sampling interval Δt is small enough such that the Nyquist frequency is higher than the highest frequency contained in the signal $x(t)$.

The discrete forms of the Fourier transform for a series of points $x_j = x(t_j)$ where $t_j = j\,\Delta t$ for $0 \le j < N$ and $\tilde{x}_k = \tilde{x}(f_k)$ where

$$f_k = \begin{cases} k\,\Delta f & 0 \le k \le N/2 \\ (N-k)\,\Delta f & N/2 < k < N \end{cases} \tag{A.108}$$

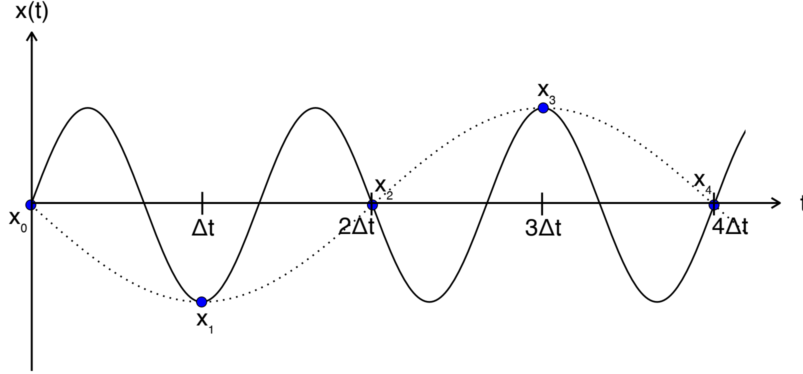

Figure A.3: An illustration of aliasing: The function $x(t) = \sin(2\pi f_0 t)$ (solid line) is sampled at intervals of $\Delta t = \frac{3}{4} f_0$ (blue points), so the Nyquist frequency $f_{\rm Ny} = \frac{2}{3} f_0 < f_0$. These points can be fit by the sinusoid $\sin(2\pi f_1 t)$ (dotted line) where $f_1 = \frac{1}{3} f_0$. The power at frequency f_0 is aliased to the lower frequency f_1.

with $\Delta f = 1/(N\,\Delta t)$ are

$$\tilde{x}_k = \Delta t \sum_{j=0}^{N-1} \mathrm{e}^{-2\pi ijk/N} x_j \tag{A.109}$$

and

$$x_j = \Delta f \sum_{k=0}^{N-1} \mathrm{e}^{+2\pi ijk/N} \tilde{x}_k. \tag{A.110}$$

We are interested in computing the *discrete Fourier transform*

$$X_k = \sum_{j=0}^{N-1} \mathrm{e}^{-2\pi ijk/N} x_j \tag{A.111}$$

and the *discrete inverse Fourier transform*

$$x_j = \frac{1}{N} \sum_{j=0}^{N-1} \mathrm{e}^{+2\pi ijk/N} X_k. \tag{A.112}$$

Note that $\tilde{x}_k = X_k\,\Delta t$. The discrete Fourier transform omits this factor of Δt. The properties listed previously for the continuous Fourier transform continue to hold for the discrete Fourier transform.

The basis functions in the Fourier series, $\exp(-2\pi ijk/N)$ are all periodic in $j \to j + N$ for all integers k, which indicates that the discrete Fourier transform is a representation of a *periodic* function $x(t)$ for which $x(t + N\Delta t) = x(t)$. If our

function is not actually periodic then some care should be taken in choosing the length of the sample. If the function has support over a limited domain in t then we can sample the function over this entire domain. Otherwise we can sample a representative portion of the function and use some method (e.g., windowing or zero-padding) to handle the fact that the samples we take will be treated as if they were taken from a periodic function.

The direct approach to computing the discrete Fourier transform would require $O(N^2)$ operations: for each of the N values of k, we would compute the sum which would require N operations. However, there is a much more efficient way of computing the discrete Fourier transform, known as the *fast Fourier transform*, that requires only $O(N \log N)$ operations. The trick to computing the discrete Fourier transform efficiently is given by the *Danielson-Lanczos lemma*,

$$\begin{aligned} X_k &= \sum_{j=0}^{N-1} \mathrm{e}^{-2\pi i jk/N} x_j \\ &= \sum_{j=0}^{N/2-1} \mathrm{e}^{-2\pi i(2j)k/N} x_{2j} + \sum_{j=0}^{N/2-1} \mathrm{e}^{-2\pi i(2j+1)k/N} x_{2j+1} \\ &= \sum_{j=0}^{N/2-1} \mathrm{e}^{-2\pi i jk/(N/2)} x_{2j} + \omega^k \sum_{j=0}^{N/2-1} \mathrm{e}^{-2\pi i k/(N/2)} x_{2j+1} \\ &= E_k + \omega^k O_k \end{aligned} \tag{A.113}$$

where

$$\omega = \mathrm{e}^{-2\pi i/N} \tag{A.114}$$

is known as a twiddle factor. We see that a Fourier transform of N points can be reduced to doing two Fourier transforms, E and O of $N/2$ even and odd points respectively. We assume that N is a power of two, so we repeat this procedure recursively $\log_2 N$ times and this results in a sum of $\log_2 N$ Fourier transforms of a single point. Of course, each one of those Fourier transforms of a single value is just that value, which is one of the values of x_j.

The range of k in E_k and O_k is $0 \le k < N/2$ while for X_k it is $0 \le k < N$. The values of X_k for $N/2 \le k < N$ can be obtained from

$$X_{k+N/2} = E_k - \omega^k O_k. \tag{A.115}$$

If we have E_k and O_k packed sequentially in a list, Y_k, of N values then we must perform a *butterfly* operation to obtain the values of X_k:

$$\begin{aligned} X_k &= Y_k + \omega^k Y_{N-k} \\ X_{N-k} &= Y_k - \omega^k Y_{N-k}. \end{aligned} \tag{A.116}$$

The Fourier transform can therefore be used to efficiently convolve or correlate functions (which are simple operations in the frequency domain). In addition, we often want to compute the power spectrum of a function. For these purposes, it is

important to make sure that the functions being convolved or correlated are zero-padded at the ends so that the periodicity imposed by the discrete Fourier transform does not cause one function to "wrap around" and spoil the result.

Another important application of the Fourier transform is in spectral analysis. The one-sided power spectral density is computed from a set of samples x_j via its discrete Fourier transform X_k as

$$P_0 = \frac{1}{N}|X_0|^2 \qquad \text{DC component} \tag{A.117a}$$

$$P_k = \frac{1}{N}(|X_k|^2 + |X_{N-k}|^2) \qquad 1 \le k < N/2 \tag{A.117b}$$

$$P_{N/2} = \frac{1}{N}|X_{N/2}|^2 \qquad \text{Nyquist component.} \tag{A.117c}$$

Here P_k is known as a *periodogram*. If the function being sampled is not periodic in the interval $N\Delta t$ then again the end effects need to be taken care of, as there will be *spectral leakage* in which the power of the continuous function at any frequency that is not exactly one of the f_k frequency bins of the discrete Fourier transform gets smeared into all of the frequency bins.

The normal technique for performing spectral analyses of aperiodic functions is to use *windowing*. A window is a smooth function of time with $0 \le t < N\,\Delta t$ that tapers off as $t \to 0$ and $t \to N\,\Delta t$. The data samples are multiplied by the window function which essentially suppresses the data near the ends of the interval. This reduces the leakage of power into frequencies that are far away from actual frequency. Some common window functions are the Hann window,

$$w_j = \frac{1}{2}\left[1 - \cos\left(\frac{2\pi j}{N-1}\right)\right], \tag{A.118}$$

the Hamming window

$$w_j = 0.54 - 0.46\cos\left(\frac{2\pi j}{N-1}\right), \tag{A.119}$$

and the Bartlett window

$$w_j = 1 - \frac{2}{N-1}\left|j - \frac{N-1}{2}\right|. \tag{A.120}$$

With data windowing, the power spectral density is given by the periodogram

$$P_0 = \frac{1}{w_{\text{ss}}}|Y_0|^2 \qquad \text{DC component} \tag{A.121}$$

$$P_k = \frac{1}{w_{\text{ss}}}(|Y_k|^2 + |Y_{N-k}|^2) \qquad 1 \le k < N/2 \tag{A.122}$$

$$P_{N/2} = \frac{1}{w_{\text{ss}}}|Y_{N/2}|^2 \qquad \text{Nyquist component.} \tag{A.123}$$

where Y_k is the discrete Fourier transform of the windowed data $y_j = w_j x_j$ and

$$w_{\text{ss}} = \sum_{j=0}^{N-1} w_j^2 \tag{A.124}$$

is the sum-squared of the window function.

If we wish to find the power spectrum of a continuous, stationary random process then the periodogram does not provide a very good estimate: the value of the periodogram at each frequency bin will have a variance that is the same as the value at that frequency. In order to reduce the variance in the power spectral estimation, a technique known as *Welch's method* is commonly used. In this method, a long stretch of data of M points is divided into K overlapping segments of N points so that segment ℓ comprises the points $x_{j+\ell N/2}$ for $0 \le j \le N-1$ where $0 \le \ell \le K-1$. Given M points and a segment length N then the number of segments is $K = 2M/N - 1$, or, conversely, the total number of points needed to form K overlapping N points is $M = N(K+1)/2$. For each of the K segments, the (windowed) periodogram is computed, and then these periodograms are averaged to obtain the power spectral density:

$$S_k = \frac{1}{K} \sum_{\ell=0}^{K-1} P_{\ell,k}. \tag{A.125}$$

The listing for `fft.py` contains routines to perform the fast Fourier transform and its inverse as well as routines to generate various kinds of window functions, the periodogram, and the power spectral density using Welch's method.

Listing A.7: Module `fft.py`

```
1  import math, cmath
2
3
4  def radix2fft(x, sign=-1.0):
5      """ Computes the FFT of a list of values.
6      Assumes the number of values is a power of 2. """
7
8      N = len(x)
9      if N == 1:   # Fourier transform of one point is that point
10         return [x[0]]
11     else:   # compute half-size ffts of even and odd data
12         X = radix2fft(x[0::2], sign)
13         X += radix2fft(x[1::2], sign)
14     omega = cmath.exp(math.copysign(2.0, sign)*cmath.pi*1j/N)
15     omegak = 1.0
16     for k in range(N//2):   # butterfly
17         tmp = omegak*X[k+N//2]
18         (X[k], X[k+N//2]) = (X[k]+tmp, X[k]-tmp)
19         omegak *= omega
20     return X
21
```

```
22
23 def fft(x, sign=-1.0):
24     """ Computes the FFT of a list of values.
25     Requires the number of values to be a power of 2. """
26
27     # make sure length of x is a power of 2
28     N = len(x)
29     if N & N-1 != 0:
30         raise ValueError('number of points must be a power of 2')
31     return radix2fft(x, sign)
32
33
34 def ifft(X, sign=1.0):
35     """ Computes the inverse FFT of a list of values.
36     Requires the number of values to be a power of 2. """
37
38     N = len(X)
39     x = fft(X, sign)
40     for j in range(N):
41         x[j] /= N
42     return x
43
44
45 def dft(x, sign=-1.0):
46     """ Computes the DFT of a list of values using the slow O(N^2)
    method. """
47
48     N = len(x)
49     y = [0.0]*N
50     for k in range(N):
51         for j in range(N):
52             y[k] += x[j]*cmath.exp(math.copysign(2.0, sign)*cmath.pi*1j*
    j*k
53                                    /float(N))
54     return y
55
56
57 def hann(N):
58     """ Create a Hann window. """
59
60     fac = 2.0*math.pi/(N-1)
61     wss = 0.0
62     w = [0.0]*N
63     for j in range(N):
64         w[j] = 0.5*(1.0-math.cos(fac*j))
65         wss += w[j]**2
66     return (w, wss)
67
68
69 def hamming(N):
```

```
70      """ Create a Hamming window. """
71
72      fac = 2.0*math.pi/(N-1)
73      wss = 0.0
74      w = [0.0]*N
75      for i in range(N):
76          w[i] = 0.54-0.46*math.cos(fac*j)
77          wss += w[j]**2
78      return (w, wss)
79
80
81  def bartlett(N):
82      """ Create a Bartlett window. """
83
84      mid = (N-1.0)/2.0
85      wss = 0.0
86      w = [0.0]*N
87      for j in range(N):
88          w[j] = 1.0-abs(j-mid)/mid
89          wss += w[j]**2
90      return (w, wss)
91
92
93  def periodogram(x, window='Rectangular'):
94      """ Computes the periodogram of a list of values.
95      Requires the number of values to be a power of 2.
96      Returns only the positive frequencies. """
97
98      N = len(x)
99      # construct window
100     if window == 'Rectangular':
101         (w, wss) = ([1.0]*N, N)
102     elif window == 'Hann':
103         (w, wss) = hann(N)
104     elif window == 'Hamming':
105         (w, wss) = hamming(N)
106     elif window == 'Bartlett':
107         (w, wss) = bartlett(N)
108     else:
109         raise ValueError('unrecognized window type')
110     # apply window to a copy of the data
111     y = [0.0]*N
112     for j in range(N):
113         y[j] = w[j]*x[j]
114     # fft windowed data and compute power
115     Y = fft(y)
116     Y[0] = Y[0].real**2/wss
117     for k in range(1, N//2):
118         Y[k] = (Y[k].real**2+Y[k].imag**2)/wss
119         Y[k] += (Y[N-k].real**2+Y[N-k].imag**2)/wss
```

```
120        Y[N//2] = Y[N//2].real**2/wss
121        return Y[:N//2+1]
122
123
124    def psdwelch(x, N, window='Rectangular'):
125        """ Computes the power spectral density of a list of values using
126        Welch's method with overlapping segments of length N.
127        Requires N to be a power of 2.
128        Returns only the positive frequencies. """
129
130        M = len(x)
131        K = 2*M//N-1  # number of segments
132        S = [0.0]*(N//2+1)  # the power spectral density
133        # compute the running mean of the power spectrum
134        for l in range(K):
135            P = periodogram(x[l*N//2:N+l*N//2], window)
136            for k in range(N/2+1):
137                S[k] = (P[k]+l*S[k])/(l+1)
138        return S
```

References

Many of the numerical techniques described above are discussed in detail in

- *Numerical Recipes: The Art of Scientific Computing* (third edition) by William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery (Cambridge University Press, 2007) ISBN 978-0-521-88068-8.

- *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables* by Milton Abramowitz and Irene Stegun (Dover, 1964) ISBN 0-486-61272-4.

Appendix B

Binary representation of numbers

Sometimes programs give surprising results for even very simple expressions. For example,

```
>>> 0.1 + 0.2
0.30000000000000004
```

Here is another example:

```
>>> 1e8 + 1e-8
100000000.00000001
>>> 1e8 + 1e-9
100000000.0
```

To understand this result it is necessary to understand how numbers are represented by a computer.

Integer numbers are naturally stored as bits in binary format. In Python, an integer can be displayed in its binary (base 2) representation as follows:

```
>>> bin(42)
'0b101010'
```

Here, the `0b` prefix indicates the number is being displayed in binary, and the following `101010` is the base 2 representation of the decimal `42`. Integer numbers are stored with a fixed number of bits, typically either 32 bits or 64 bits depending on the type of computer. Python is also capable of storing much larger integers in variable width representations. For example, Python can compute a googol, 10^{100}:

```
>>> 10**100
100000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000L
```

The trailing `L` indicates that this is a variable width long representation. Although Python handles variable width integers, most other computer languages support only fixed width integers. The smallest and largest integers that can be stored in a 64 bit signed integer type are $-2^{63} = -9\,223\,372\,036\,854\,775\,808$ and $2^{63} - 1 = 9\,223\,372\,036\,854\,775\,807$.

Real numbers are approximated in a *floating point* representation. The floating point number is composed of a significand, having some fixed number of digits, and an exponent,

$$\text{signficand} \times \text{base}^{\text{exponent}}$$

so, for example, the number $\frac{617}{500}$ can be written

$$\frac{617}{500} = 1.234 = 1234 \times 10^{-3}$$

so its floating point significand is `1234` and its (base 10) exponent is `-3`. In this case, the real number can be exactly represented with a four decimal digit significand. Because the significand has a finite precision, not all real numbers can be exactly represented. Irrational numbers can never be exactly represented, and some rational numbers cannot be exactly represented in a given base. For example, $\frac{1}{9}$ cannot be exactly represented in base 10, but in base 3 it is just 1×3^{-2}.

On computers, the significand and exponent are stored as a binary integer with a certain number of digits and the base is 2. The most common way that real numbers are approximated on a computer is as *double precision* IEEE 754 floating point numbers. In this case, the significand has 53 bits of precision. (Only 52 bits of the significand actually need to be stored because the first bit will always be `1`.) The exponent has 11 bits and a final bit indicating the sign of the number brings the total number of bits in a double precision floating point number to 64.

Python's `bin` routine cannot be used to show the binary representation of a floating point number. The following is a routine that will:

```
def binary(x, precision=53):
    s = '-' if x < 0 else '' # write the sign
    ip, fp = abs(x) // 1, abs(x) % 1 # integer and fractional parts
    s += bin(int(ip)) + '.' # write the integer part and radix point
    if ip != 0:
        precision -= len(s) - 2
    # write the fractional part
    for i in range(precision):
        s += str(int(2*fp))
        fp = (2*fp) % 1
    return s
```

Now we can get the binary representation of numbers such as 0.1 and 0.2:

```
>>> binary(0.1)
'0b0.00011001100110011001100110011001100110011001100110011'
>>> binary(0.2)
'0b0.00110011001100110011001100110011001100110011001100110'
```

It is clear that 0.1 and 0.2 are not terminating in base 2; they are $0.0\overline{0011}_2$ and $0.\overline{0011}_2$ respectively. Because the computer represents these numbers with finite precision, the representation is not exact. Rather than 0.1 and 0.2, the computer uses the values

$$\frac{900719925474099}{9007199254740992} \quad \text{and} \quad \frac{900719925474099}{4503599627370496}.$$

The result of adding these numbers is seen in the following:

```
>>> binary(0.1 + 0.2)
'0b0.0100110011001100110011001100110011001100110011001101010'
```

Notice that there is a small error at the end: the last two digits should have been `01` rather than `10` as can be seen as follows:

```
>>> binary(0.3)
'0b0.0100110011001100110011001100110011001100110011001100110011001'
```

The reason for the error is that that the numbers 0.1 and 0.2 were only approximately represented to begin with, and errors accumulate as rounding off is performed multiple times.

An immediate consequence of roundoff error is that one needs to be careful when comparing floating point numbers:

```
>>> 0.1 + 0.2 == 0.3
False
```

Instead, use a function similar to this one to see if two floating point numbers are close:

```
def isclose(x, y, rtol=1e-5, atol=1e-8):
    return abs(x - y) <= (atol + 0.5 * rtol * abs(x + y))
```

Here, `atol` is the absolute tolerance and `rtol` is the relative tolerance. A routine similar to this is provided in `numpy`.

The finite precision of the floating point numbers representation in a computer can lead to catastrophic cancellation in subtracting numbers that are very similar. For example, consider the function

$$f(x) = \frac{1 - \cos x}{x^2}$$

which could be implemented in python as

```
import math
def f(x):
    return (1.0 - math.cos(x)) / x**2
```

For small values of x, one expects $f(x) \approx \frac{1}{2}$. However, if the function is evaluated at $x = 1.1 \times 10^{-8}$,

```
>>> f(1.1e-8)
0.917539689773
```

which is quite far from the true value. A more numerically stable implementation would be

```
import math
def f(x):
    return 2.0 * math.sin(0.5 * x)**2 / x**2
```

Now one obtains the more satisfying result

```
>>> f(1.1e-8)
0.5
```

The precision of a computer's floating point representation is called the *machine epsilon*. It is the smallest number that can be added to `1.0` that gives a distinctly different number. It can be determined as follows:

```
epsilon = 1.0
while 1.0 + 0.5 * epsilon != 1.0:
        epsilon *= 0.5
print repr(epsilon)
```

The result is `2.220446049250313e-16` which is 2^{-52}. In addition to the precision of a computer's floating point representation, there are limits on the largest number that can be represented, and the tiniest number that is usable. The minimum 'normal' positive double-precision floating point number is 2^{-1022} or `2.2250738585072014e-308` and the maximum double-precision floating point number is $(1 + (1 - 2^{-52})) \times 2^{1023}$ or `1.7976931348623157e+308`. Attempting to create a larger number can result in an overflow error:

```
>>> 2.0**1024
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
OverflowError: (34, 'Result too large')
```

There are some special floating point numbers. Positive and negative infinity are floating point numbers that are always greater than or less than any representable floating point number respectively. In Python, these numbers can be obtained as `float('inf')` and `float('-inf')`. The final number is called *not a number*, or NaN, and results from impossible floating point operations:

```
>>> 0 * float('inf')
nan
>>> float('inf') + float('-inf')
nan
```

The NaN is a useful as an invalid floating point value. In Python it can be obtained with `float('nan')`.

Exercise B.1 Consider the function

$$f(x) = \frac{1}{\sin^2 x} - \frac{1}{x^2}.$$

a) Use calculus to determine the exact value of

$$\lim_{x \to 0} f(x).$$

b) Write a program to evaluate $f(x)$, and evaluate it for $x = 10^{-1}$, $x = 10^{-2}$, $x = 10^{-3}$, $x = 10^{-4}$, $x = 10^{-5}$, $x = 10^{-6}$, $x = 10^{-7}$, $x = 10^{-8}$. Which of these gives the closest approximation to your result from part (a)?

c) Explain the results of part (b).

Further reading

More details about floating point numbers and the IEEE 754 standard can be found in these references:

- *Floating Point Arithmetic: Issues and Limitations* The Python Tutorial `https://docs.python.org/2/tutorial/floatingpoint.html`
- *What Every Computer Scientist Should Know About Floating-Point Arithmetic* by David Goldberg (1991) `http://perso.ens-lyon.fr/jean-michel.muller/goldberg.pdf`
- *IEEE Standard for Floating-Point Arithmatic* (2008) `http://ieeexplore.ieee.org/servlet/opac?punumber=4610933`

Index

absolute error, 41
absolute magnitude, 151
acceptance rate, 161
adaptive scheme, 40
advection equation, 50
aliasing, 202
anti-correlated, 131
areal velocity, 31

bisection method, 20, 175
boundary value problems, 47
bracket, 20, 175
bracketing triplet, 178
butterfly, 204

Cash-Karp parameters, 41
Cayley's form, 73
chaos, 29
chi-square fitting, 147
chi-squared distribution, 141
Cholesky decomposition, 154
coarse graining, 102
conditional probability, 159
confidence level, 154
convolution, 201
correlated, 131
correlation, 201
correlation coefficient, 155
Courant condition, 55
Crank-Nicolson method, 73
critical exponent, 113
Crout's algorithm, 172
cumulative distribution, 128
Curie temperature, 111

Danielson-Lanczos lemma, 204
design matrix, 152
detailed balance, 160
diffusion constant, 102
Dirichlet boundary conditions, 48
discrete Fourier transform, 203
discrete inverse Fourier transform, 203
distance modulus, 151
distribution, 105
double precision, 212

eccentric anomaly, 32
eccentricity, 32
eigenenergies, 77
eigenfunctions, 77
eigenstates, 77
eigenvalues, 77
elliptic equations, 47
empirical distribution function, 134
entropy, 105
error function, 129
Euler beta function, 132
Euler's method, 13
Euler-Maclaurin formula, 188
evidence, 159
exchange energy, 111
extended Simpson's rule, 189

fast Fourier transform, 204
ferromagnetic phase, 112
floating point, 212
flux-conservative, 50
forward time, centered space, 52
frequentist, 159
full width at half maximum, 127

gamma function, 132
Gauss-Seidel method, 93
Gaussian distribution, 129

Gaussian quadratures, 191
general linear least squares, 152
global error, 17
golden ratio, 179
goodness of fit, 148

heat capacity, 114
Heaviside step function, 135
heliocentric gravitational constant, 35
Hermite polynomial, 77
hyperbolic equations, 47

implicit differencing scheme, 68
improper prior, 162
incomplete beta function, 132
incomplete gamma function, 141
initial value problems, 47
interquartile range, 127
Ising model, 111

Jacobi's method, 90

K-S test, 134
Kepler problem, 30
Kepler's equation, 32
Kepler's first law, 31
Kepler's second law, 31
Kepler's third law, 32
Kolmogorov-Smirnov test, 134

Lagrange's interpolation formula, 184
Laplace-Runge-Lenz vector, 32
Lax method, 55
leapfrog method, 58
Legendre polynomials, 194
Legendre's differential equation, 194
likelihood function, 159
likelihood ratio, 162
linear correlation coefficient, 131
linear interpolation, 184
linear regression, 147
local error, 17
LU decomposition, 169

machine epsilon, 214
magnetic susceptibility, 114
marginalized likelihood, 159
Markov chain Monte Carlo, 160
mean, 125
mean anomaly, 34
mean field theory, 112
measures of central tendency, 125
median, 125
Metropolis algorithm, 114
Metropolis-Hastings algorithm, 160
midpoint method, 186
midrange, 126
mode, 125
Monte Carlo integration, 196
Monte Carlo methods, 101
Moore-Penrose inverse, 155
multimodal, 126
multiplicity of states, 105

Neville's algorithm, 185
Newton's method, 175
normal distribution, 129
normal equations, 153
not a number, 214
null hypothesis, 131
Nyquist frequency, 202

occupation numbers, 105
operator splitting, 83
order statistics, 125
over-relaxation parameter, 93

p-value, 132
parabolic equations, 47
paramagnetic phase, 111
parsecs, 151
Parseval's theorem, 202
partition function, 106
Pearson's chi-squared test, 134
Pearson's r, 131
periodogram, 205
polynomial interpolation, 184
posterior probability, 159
prior probability, 159
probability density function, 128
proposal distribution, 160
pseudoinverse, 155

quadrature, 186

quantile function, 128

random walk, 101
rectangular method, 186
reflective boundary conditions, 51
relative error, 41
relaxation method, 89
Richardson extrapolation, 190
Riemann sum, 186
Rodrigues' formula, 194
Romberg integration, 191
root finding, 20
Runge-Kutta method, 36

sampling theorem, 202
second-order phase transition, 113
separation of variables, 49
shooting method, 19
simulated annealing, 179
singular value decomposition, 155
spectral leakage, 205
spectral radius, 92
standard deviation, 127
standard scores, 129
state, 105
statistical range, 126
step doubling, 40
Stirling's approximation, 107
successive over-relaxation, 93

thermal diffusivity, 64
transpose, 153
trapezoid method, 187
tridiagonal matrix, 69, 172
true anomaly, 32
truncation error, 13

uncorrelated, 131

variance, 128
variational method, 119
vis-viva equation, 34
von Neumann stability analysis, 53

Welch's method, 206
windowing, 205