Scientific Programming for Kids

a teacher's manual for charismatic hackers

lessons 1-4: fundamental techniques and examples

(work in progress)

Mark Galassi
Los Alamos National Laboratory
and
Warehouse 21, Santa Fe, New Mexico

`mark@galassi.org`

June 15, 2016

Contents

List of Figures

Motivation and plan

I cannot imagine a career more wonderful than that of a scientist.

The day-to-day work in science today involves using computers at all times. Scientists who master their computers and can program them with agility are the ones who enjoy the job the most and are often in great demand: they can carry out unique new research.

I have developed a series of lessons on scientific computing, aimed at kids who have already taken my "Serious Programming For Kids" course (Galassi 2015). This booklet covers lessons 1-4. I have two goals with these lessons: *(a)* introduce the tools and tricks for scientific computing, and *(b)* take a tour of diverse scientific problems that demonstrate "realy interesting" things you can do with some programming knowledge.

The course teaches scientific computing using Python on the GNU/Linux operating system. There are other possible choices of programming language and operating system, and some of them are adequate, but there are specific reasons for which I chose Python and GNU/Linux. Some are those given in the "Serious Programming for Kids" teacher's manual, but here are some other reasons which are specific to scientific work:

- Scientific software often matures into sophisticated programs which need to be executed on production computers and in a reproducible manner. For this the use of a free/open-source operating system and language interpreter are crucial.

- Much scientific infrastructure is available as an integral part of the GNU/Linux distributions. For example, on a current Debian GNU/Linux or Ubuntu distribution you will find that the GNU Scientific Library, astropy, scipy, a remarkable number of R science packages, and much much more are "just there" as part of the operating system. This comes in part from the fact that the GNU/Linux operating system is developed by hackers for hackers: programming is a seamless part of such systems.

- Python spread rapidly soon after its initial development. Thanks to some key early developers being part of physics, astronomy and biology research groups, it was rapidly adopted by the scientific community. The result is a vast collection of scientific libraries.
- Many research projects have very long lives, and the software is used for years after it is first written. My opinion, and that of many who observe the business of scientific computing, is that programs written in Python on a GNU/Linux system will be stable[1]
- Reproducibility again: using proprietary software in scientific research makes it impossible to reproduce or verify a result.
- Reproducibility and verifiability also dictate that scientific software should be able to run in *batch mode*, rather than through a graphical user interface (GUI).*A GUI is not necessarily a bad thing, but after initial exploration of data with a GUI, the scientist needs to then generate a batch program to reproduce the results.*

Notes for teachers

This is a teacher's manual for the course. If you want to follow my format, I recommend lecturing at a blackboard (nowadays probably a whiteboard) with a printout of this in hand for code samples.

For the scientific course it would also be good to have a computer set up on a projector to sometimes show plots, as well as a web site with all the code samples available for rapid download: most of the examples are meant to be typed in by the students during the lectures, since typing them in is part of the process of learning the material, but it might be necessary sometimes to "just grab the `.py` file."

One special case will be the use of long-ish URLs. Some of the examples involve writing programs that download data sets from the web (temperature, population, audio...) Or we might use `wget` on the command line to get those sets. Writing the full URL on the board, or dictating it, or putting it in a slide will not work. Here are a couple of ways to get that URL to the students without having them type it: *(a)* have them do a web search with well-chosen keywords and then get the URL

[1]Programs written in the C programming language on a GNU/Linux system will be even more stable, thanks to the maturity and stability of the C standard. C is also a delightful and powerful language, but it is not in the scope of what I teach to younger kids.

from their browser, or *(b)* the instructor puts a text file with the URLs in an easy place her/his web site. The first solution is has a nice instructional side-effect (show kids how to search for data on the web) but also has the drawback that over time the search results might change.

The lecturing style should be one of quickly getting a juicy example up on their screens: something that gives visible results for the students. Then step back a bit to make sure they understood how we got to it, and then quickly on to the next example.

This is hard work for the students: I have developed this course to include serious material they might otherwise not learn until college, so I often ask the students to "suspend their not understanding"[2] and just latch on to *one or two things they can remember*. For example I introduce Fourier Analysis (Chapter 4), and when I give that lecture I frequently repeat "remember: it is OK to not understand most of this, but repeat after me the one thing I want you to understand: *all these signals look like wild jumbles, but they are made up of simple waves which let us understand part of their musical nature.*"

In broad strokes you can think of two main categories of scientific computing effort: analyzing data from experiments, and simulating your own physical situation with a computer program that generates fake (but, we hope, realistic) experimental data. We will look at both of these types, and introduce the words: *experiments* and *simulation* as we go through the examples.

The way in which kids approach computers today allows them to not understand some concepts which are very important for scientific programming (and in fact any kind of programming). Because of this we must first get comformtable with the following concepts:

- What is a data file.
- How to plot a data file.
- How to write a program which takes a data file, does some processing of the data, and writes out another file with the processed data.

Once we have these skills we can:

- Tell the story of that plot.

[2]A pun on Coleridge's "suspension of disbelief" – with topics of great complexity it is important for students to be flexible about temporarily accepting a building block that they don't undersand so that they can keep with the flow.

- Generate simulated data.
- Retrieve data from online sources.
- Record data from an experiment.
- Analyze data to go *beyond that initial story.*

This first book has four 1-hour lessons which will give exposure to these areas while also providing little nuggets of data analysis which can be applied to diverse problems[3].

Reproducibility and how to build this book

This book is available to you under the terms of the GNU Free Documentation License (GFDL; see Appendix A). The license allows you to adapt it to your own needs and to redistribute modified copies if you should need to.

You can get a copy of the *source material* for this book from the Bitbucket hosting site. The project is at `https://bitbucket.org/markgalassi/hackingcamp-teacher-manual` and you can clone and build the book with:

```
$ hg clone https://markgalassi@bitbucket.org/markgalassi/hackingcamp-teacher-manual
$ cd hackingcamp-teacher-manual/teacher-manual
$ make
```

oafter which you can view `scientific-computing-1to4.pdf` with your favorite PDF viewer.

To make the examples in this book easily reproducible, and to insert their plots and code snippets automatically, I provide a program called `make-book-example.py` which runs all the programs and makes all the plots needed by the book. Note that this showcases an important advanced concept in scientific computing: you need to build all the plots for your papers with an *automatic and reproducible script.*

The book is generated by running:

```
$ ./make-book-example.py scientific-computing-1to4.tex
$ pdflatex  scientific -computing-1to4.tex
## and then the whole lytany of running:
$ biber  scientific -computing-1to4.tex
$ pdflatex  scientific -computing-1to4.tex
$ pdflatex  scientific -computing-1to4.tex
```

or more concisely:

[3]Note that I have not yet taught the course to kids - I am writing this book in preparation for it - so I might revise the “1-hour” estimate of how long it takes!

```
$ ./make-book-example.py scientific-computing-1to4.tex
$ latexmk -pdf  scientific -computing-1to4.tex
```

or even more concisely just type `make` to do the whole thing, and you can examine the book's `Makefile` to see how it is done.

Acknowledgements

Thanks to David Palmer and Laura Fortunato for discussing this curriculum with me in great detail. Thanks to Antoni Galassi for listening carefully as I tested my explanations of this material on him.

Lesson 1

Starting out: data files and first plots

1.1 Very first data plots with gnuplot

Our first goal is to become comfortable with data files and with plotting. We first get the students to renew their acquaintance with creating files with an editor and make a file with some hand-crafted data.

Use your favorite editor (possibly `emacs` for those who have taken my previous course, but `vi` or `gedit` should also work) to open a file called `simpledata.dat`

Enter two columns of simple data into this file. For example:

```
-3    2.7
-2.5  2.1
-2    2.0
-1.5  2.2
-1    2.7
-0.5  2.8
0     2.9
0.5   3.1
1.0   2.8
1.5   2.3
2.0   1.8
```

Then save it, and enter `gnuplot` to plot this data:

```
$ gnuplot
gnuplot> plot 'simpledata.dat'
```

then have the students plot the data with slightly different options in gnuplot:

```
$ gnuplot
gnuplot> plot 'simpledata.dat' with lines
gnuplot> set xlabel 'this is the "x" axis'
gnuplot> set ylabel 'this is the "y" axis'
gnuplot> plot 'simpledata.dat' with linespoints
```

Note that we want to give early hints to how you can make this automatic and reproducible, so we will also give an example of automatically making a PDF file and including into a document. We should make the students input this file, but on the projector we can show that Figure 1.1 is generated by the gnuplot script `simpledata.gp` running on the file `simpledata.dat`

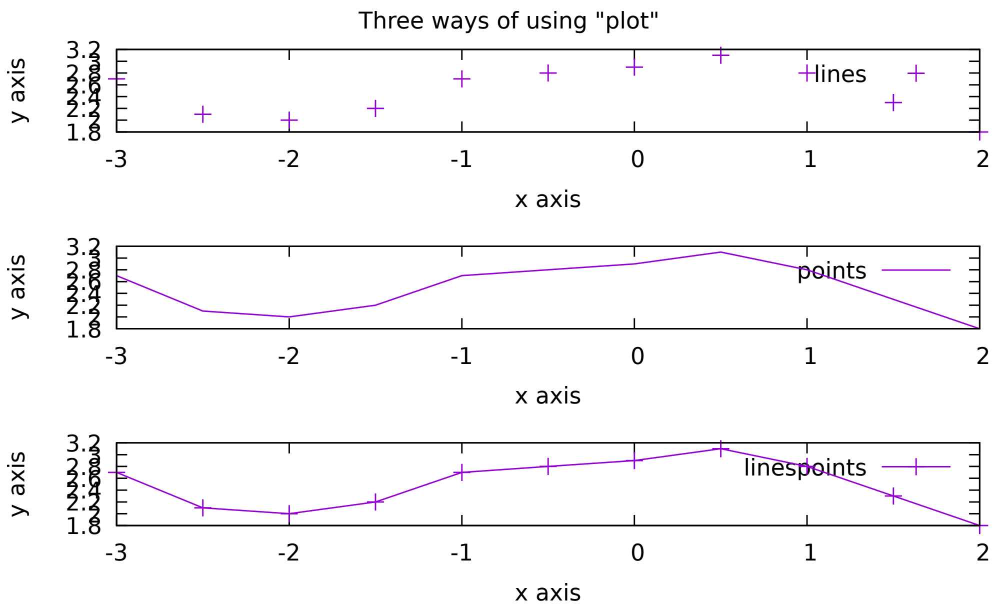

Figure 1.1: Plot generated by running gnuplot with “`gnuplot simpledata.gp`”

gnuplot instructions

```
set xlabel 'x axis'
set ylabel 'y axis'
set multiplot layout 3,1 title 'Three ways of using "plot"'
plot 'simpledata.dat' using 1:2 title 'lines'
plot 'simpledata.dat' using 1:2 with lines title 'points'
plot 'simpledata.dat' using 1:2 with linespoints title 'linespoints'
```

1.2 Plotting *functions* with gnuplot

More examples of using gnuplot. We don't assume knowledge of trigonometry from younger students, so we tell the story as "this is the sin function, which you will learn about some day; it plots these waves."

On the other hand the polynomial $y = x^3 - 3x$ should be within their reach: I might call on the class to tell me "what's -10^3 and -2^3, 2^3, and 10^3 – this establishes that the plot goes down on the left hand side, and up on the right hand side. The interesting play in the middle can be narrated by showing that $(1/2)^3$ is smaller than $3 \times (1/2)$, so that the negative term dominates briefly (Figure 1.2).

```
gnuplot> plot sin(x)
gnuplot> plot x*x*x -3*x
gnuplot> plot x**3 -3*x
## note that this last one did show a dip in the middle,
## but zooming in on the range from -3 to 3 shows the
## interesting features in the middle of the plot. This
## plot has two flat points (one local maximum and one
## local minimum), rather than one saddle point.
gnuplot> plot [-3:3] x**3 -3*x
```

(advanced material)

If I am getting a good mathematical vibe from the classroom I will briefly step into calculus territory and ask students to predict the local max and min for $y = x^3 - 3x$. I will then quickly show them that

$$\frac{d(x^3 - 3x)}{dx} = 3x^2 - 3$$

and using the quadratic formula

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

we get $\pm 6/6 = \pm 1$, which is exactly where the local max and min are in Figure 1.2.

You can now set the grid in the plot and see if this calculation matches the plot:

```
gnuplot> set grid
gnuplot> plot [-3:3] x**3 -3*x
## while we're at it also show the derivative:
```

```
gnuplot> replot 3*x**2 -3
```

The two plots, superimposed in Figure 1.2, show that where the derivative function ($3x^2 - 3$) is zero, the original function ($x^3 - 3x$) has its flat point. This can be presented, especially to older kids, in a rapid way that conveys "you don't have to understand this, but if you have heard about slopes then note that the second curve shows the slope of the first one…" For the younger kids we can emphasize that the figure shows two functions and looks intriguing.

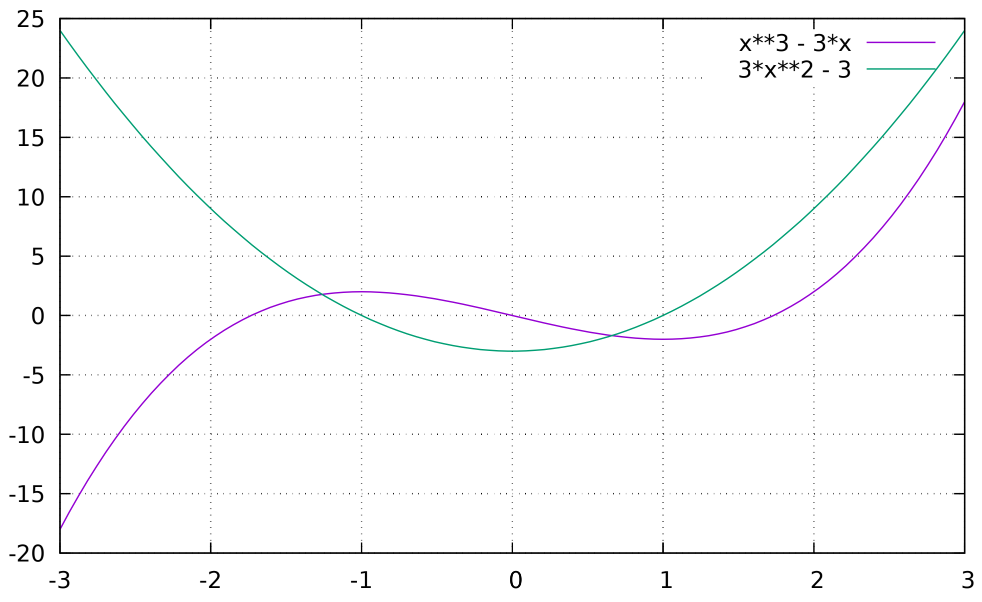

Figure 1.2: Plot of a simple function and its derivative. This plot was generated by these instructions:

gnuplot instructions

```
set grid
## plot both function and derivative
plot [-3:3] x**3 - 3*x, 3*x**2 - 3
```

Visualizing functions like this is cool and it can be useful during the exploratory phase of a research project, but it seldom comes up in the bulk of the work and in the final scientific write-up. We will now move on to tasks which come up quite frequently.

1.3 Reading and writing files, in brief

First let us make sure we know a couple of shell commands to look at a file. Here I usually will take a portion of the board and write a boxed inset "cheat sheet" with some useful shell commands.[1]

Since we already have a file called `simpledata.dat` which we created earlier, let us look at three shell commands that give us a quick glance at what's in that file: `head`, `tail` and `less`.

```
$ head simpledata.dat
$ tail simpledata.dat
$ less simpledata.dat
## (when using less be sure to quit with 'q')
```

These are simple ways to peek at a file, and will work with any text file. You should always remember these commands.

Next we will look at how to read a file in a Python program. This is a crucial pattern and we will use it a lot. Type in the program `simple-reader.py` and run it to see what happens.

```
#! /usr/bin/env python3

"""show a simple paradigm for reading a file with two columns of
data"""

def main():
    fname = 'simpledata.dat' # the file we wrote out by hand
    dataset = read_file(fname)
    print('I just read file %s with %d' % (fname, len(dataset)))
    print('I will now print the first 10 lines')
    for i in range(10):
        print(dataset[i])

def read_file(fname):
    dataset = []
    f = open(fname, 'r')
    for line in f.readlines():
        words = line.split()
        x = float(words[0])
        y = float(words[1])
        dataset.append((x, y))
```

[1]In the introductory course I have insets on the board with shell commands, `emacs` keybindings, and some Python commands. The `emacs` keybindings are especially important since the students have not necessarily done the full tutorial.

```
    f.close()
    return dataset

if __name__ == '__main__':
    main()
```

Listing 1.1: Reads a simple set of 2-column data: **simple-reader.py**

Remember that after writing and saving the program you do the following to make it executable and then run it:

```
$ chmod +x simple-reader.py
$ ./simple-reader.py
```

Finally let us see how to write files to disk. We will extend `simple-reader.py` to do an easy manipulation of the file `simpledata.dat` and then write it back out to a new file `simpledata.dat.sums`. This new program will be called `simple-writer.py`, so we need to copy it first:

```
$ cp simple-reader.py simple-writer.py
```

and edit the new file.

```
#! /usr/bin/env python3

"""show a simple paradigm for writing a file after reading it and
adding some content to it"""

def main():
    fname = 'simpledata.dat' # the file we wrote out by hand
    dataset = read_file(fname)
    print('I just read file %s with %d' % (fname, len(dataset)))
    print('I will now print the first 10 lines')
    for i in range(10):
        print(dataset[i])
    print('I will now modify the data')
    summed_data = append_sums(dataset)
    write_file(fname + '.sums', summed_data)

def read_file(fname):
    dataset = []
    f = open(fname, 'r')
    for line in f.readlines():
        words = line.split()
        x = float(words[0])
        y = float(words[1])
        dataset.append((x, y))
    return dataset

def append_sums(dataset):
    sum_y = 0
    summed_data = []
    for pair in dataset:
        sum_y = sum_y + pair[1]
        triplet = (pair[0], pair[1], sum_y)
        summed_data.append(triplet)
    return summed_data

def write_file(fname, dataset):
    f = open(fname, 'w')
    for triplet in dataset:
        f.write('%g  %g   %g\n' % triplet)
    f.close()

if __name__ == '__main__':
    main()
```

Listing 1.2: Reads a simple set of 2-column data and sums the second column and then writes out line with (x y sumy): **simple-writer.py**

At this point we are comfortable with basic programming patterns for reading and writing data. This is very important: this kind of file manipulation is one of the big steps in becoming a confident programmer. This is a good time to make the kids familiar with the terminology "input/output" (I/O).

1.4 Generating our own data to plot

Now we look at an example of writing a python program to generate some data which we will then plot. Initially this just feels like silly data: it calculates the *sin* function for many opints and prints out the values. There is a reason to start with this very simple model: it will allow us (in Section 4.3) to give clear cut examples of deeper data anlysis. We will not be lazy: later on (in Section 4.4) we will look at real signals instead of the toy ones.

Start with the `python3` command line and let us see the few lines of code that generate *sin* wave data:

```
#! /usr/bin/env python3
import math

for i in range(200):
    x = i/10.0
    signal = math.sin(x)
    print('%g   %g' % (x, signal))
```

Listing 1.3: `make-simple-wave.py` - simple sin wave generator

You can see the plot in Figure 1.3,

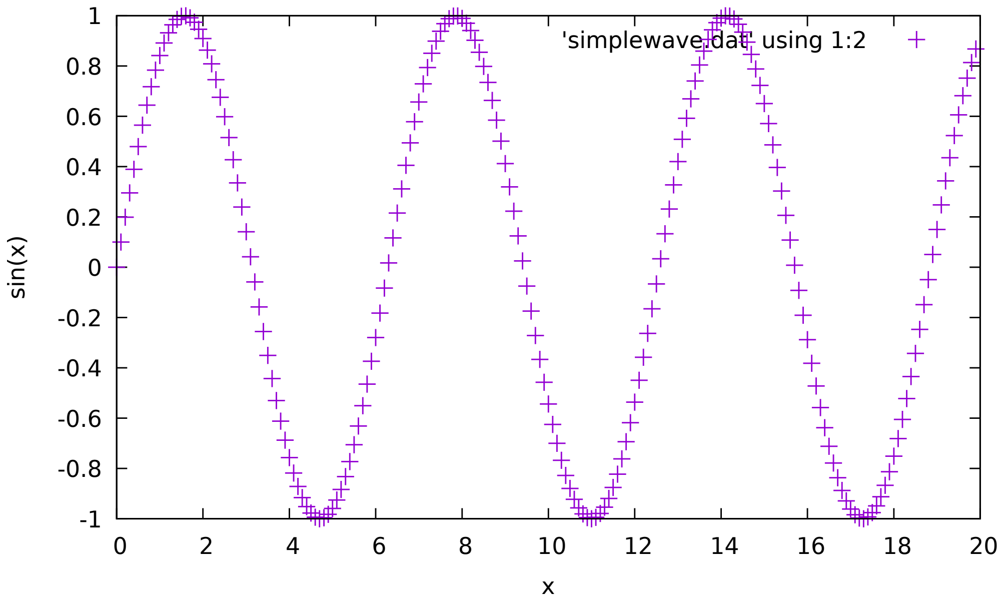

Figure 1.3: Plot generated by the following: run the commands at the shell to generate the data, then plot the data with with "`gnuplot simplewave.gp`"

commands to generate data

```
./make-simple-wave.py > simplewave.dat
```

gnuplot instructions

```
set xlabel 'x'
set ylabel 'sin(x)'
plot 'simplewave.dat' using 1:2
```

Comments on this:

- This just prints values to the terminal, so we limited it to 30 values of `x`. When we write it to a file will do much more.
- The classic `for` loop generates a sequence of integers, which does not do well at all for plotting: we want to space our `x` values much more tightly to get a nice plot, so we divide the integer `i` by 10.0 to get finely spaced values of `x`.

Let us now put this into a file called `generate-sin-first-try.py`

```python
#! /usr/bin/env python3

import math

def main():
    out_fname = 'sin_wave.dat'
    f = open(out_fname, 'w')
    for i in range(700):
        x = i/100.0
        signal = math.sin(x)
        f.write('%g   %g\n' % (x, signal))
    f.close()
    print('finished writing file %s' % out_fname)

if __name__ == '__main__':
    main()
```

Listing 1.4: First stab at python program: `generate-sin-first-try.py`

We run the program and do a quick scan of its output with:

```
$ python3 generate-sin-first-try.py
$ ls -l sin_wave.dat
$ head sin_wave.dat
$ tail sin_vave.dat
```

and when we have seen the first and last few lines of the output file we realize we can plot it (after going back to our gnuplot window) with:

```
$ gnuplot
gnuplot> plot 'sin_wave.dat' using 1:2 with lines
```

Describing what this program does is rather straightforward, and it can be compared to the gnuplot instruction `plot sin(x)`.

At this point, at the blackboard, I will suggest to the students a couple of edits they "should have already tried on their own":

- See what happens if we don't divide $i/100.0$: try both `x = i` (for this use `range(7)`) and some intermediate value `x = i/4.0` (for this use `range(28)`). Note the loss of resolution.

- Use python's `sys.argv` to take the file name as a command line argument.

We then show a cleaner version of this program which adds some comments, makes clear what the *sin* period is, and uses some robust python proramming paradigms (especially `with ... as ...`), which at this time students should take on faith is the "right way" of opening files.

```
#! /usr/bin/env python3

"""
Generate samples of a simple sin() wave and save them to a
file. The file has two columns of data separated by white
space. To run the program and plot its output try:
$ python3 generate-sin-cleaner.py 'sin_output.dat'
$ gnuplot
gnuplot> plot 'sin_output.dat' using 1:2 with lines
"""

import math
import sys

def main():
    out_fname = 'sin_wave.dat' # default output filename
    if len(sys.argv) == 2:
        out_fname = sys.argv[1]
    elif len(sys.argv) > 2:
        print('error: program takes at most one argument')
        sys.exit(1)

    with open(out_fname, 'w') as f:
        for i in range(700):
            x = i/100.0
            signal = math.sin(x)
            f.write('%g   %g\n' % (x, signal))
        print('finished writing file %s' % out_fname)

if __name__ == '__main__':
    main()
```

Listing 1.5: Cleaner version: `generate-sin-cleaner.py`

Note that the comments also tell you how to run a program and visualize the output. This is an important detail, even for small programs.

In Section 4.3 on we will revisit this simple program and make it generate more complex waveforms.

1.5 The broad landscape of plotting software

Now that we have seen some examples, let us talk broadly about plotting software, since you will soon be bombarded with people telling you about their favorites.

By now we know well that in the free software world there is often a dizzying variety of tools to do any task, with fans advocating each approach. Gnuplot is full-featured, stable, actively maintained and ubiquitous so I have chosen it, there are several other valid choices.

There are at least three main categories of plotting tools: *(a)* the "just a plotting program" kind, *(b)* the "plotting program with some data analysis that grew into a full programming language", *(c)* the "plotting library for a well-established programming language". Gnuplot is clearly one of the first, R and Octave the second, Python with Matplotlib and Cern's Root are examples of the third.

Often it comes down to where a particular scientist did her early research work: a boss will tell you to "use this tool because it's what I use". I recommend forming a broad knowlege of scientific tools so that you can use *the most appropriate tool* instead of *the tool that makes your boss comfortable.* You will often find that astrophysicists often use Python with matplotlib, particle physicists use Cern's Root, biologists use R or Python, social scientists who do much statistical work use R. You shoud always know the tool your community uses (if it's a free software tool), as well as some others which might be more appropriate.

There are many proprietary plotting packages. I advocate against the use of proprietary software, and it is certainly unacceptable to do science with proprietary tools, but I will mention a couple of packages so that when you come across users you will be able to categorize and compare them and offer an effective free software approach to the same problem.

CricketGraph, often used in the 1990s, was a light-weight plotting program, later supplanted by KaleidaGraph. I would recommend gnuplot as a good way of doing what those programs did.

Matlab and IDL are simple plotting and data analysis programs that grew out of control and added an ad-hoc programming language to the package. The languages were never meant for large software applications, but are often used to write very large programs. It is interesting to note that these programs always start with the stated intention of not requiring a scientist to know how to program, but they end up channeling scientists into using an ill-designed language for large programs.

Matlab and IDL programs can be written in a much cleaner way using Python for the programming parts, and the matplotlib plotting library for graphics.

There is a final outlier in the proprietary data analysis world, which is the "using a spreadsheet to do data analysis" approach, often with the proprietary Excel spreadsheet. There is no saving grace to this approach: apart from technical concerns with the validity of the numerical subroutines, there is also the complete lack of reproducibility of a person moving a mouse around over cells. One of the most embarrassing cases of incorrect analysis was in a much-cited Economics paper about debt rations in European countries Reinhart and Rogoff 2010. The analysis was done with an Excel spreadsheet, and some readers concluded that the authors selected the wrong range of data with a mouse movement. There is no reproducibility when mouse movements are part of the data analysis. The economics article was disgraced because of the faulty analysis as well as other problems with their methodology.

1.6 Data formats

In Sections 1.1 and 1.3 we saw the simplest examples of data files: columns of numbers separated by white space. These are the simplest to work with, and if your files are smaller than about 10 megabytes you should always treat yourself to that simplicity. This format is often called "whitespace separated ascii" or names similar to that.

Often you will find that the columns of data are separated by commas. This format is called "comma separated values" (csv) and the files often end with the `.csv` extension. The format has been around almost half a century. It has some advantages over the whitespace separated columns and is used by almost all spreadsheet programs as an import/export format.

Sometimes files are in a variety of *binary formats*. We will not deal with these at this time, since we are not yet working with very big files, but later on in Crefsubsec:white-noise we will show how to convert mp3 files to an ascii format which is easily read by our programs and by gnuplot.

1.7 Population data from the web

Our goals here are to:

- Automate fetching of data sets from the web.

- Look at a plot in a few different ways to get a narrative out of it.

We will start by looking at the population history of the whole world. When I discuss this with students I often ask "what do you think the population of the world is today?" (then you can have them search the web for "world population clock", which will take them to `http://www.worldometers.info/world-population/`).

Then ask "what do you think the world population was in 1914? And 1923? And 1776? And 1066? And in the early and late Roman empire? And in the Age of Pericles?

Let us search for

`world population growth`

and we will come to this web site: `https://ourworldindata.org/world-population-growth/` and if we go down a bit further we will see a link to download the annual world population data.

We will *not* click on the link. Instead we will use the program `wget` to download it automatically[2]:

```
$ wget http://ourworldindata.org/roser/graphs/[...]/....csv -O world-pop.csv
```

Note that this is a very long URL, but students can get it as a result of their search, so nobody has to type the full thing in.

Once they have the file downloaded they can look at the data with:

```
$ less world-pop.csv
```

and will quickly see that it is slightly different from the data we have seen so far. The columns of data are separated by *commas* instead of spaces. This type of file format is called *comma-separated-value* format and is quite common. Our plotting program, `gnuplot`, works with space-separated columns by default, so there are two tricks to plot the file. Either use the cool program `sed` to change the commas into spaces:

```
$ sed 's/,/ /g' world-pop.csv > world-pop.dat
$ gnuplot
gnuplot> plot 'world-pop.dat' using 1:3 with linespoints
```

or tell gnuplot to use a comma as a column separator:

[2]The full URL is `http://ourworldindata.org/roser/graphs/WorldPopulationAnnual12000years_interpolated_HYDEandUN/WorldPopulationAnnual12000years_interpolated_HYDEandUN.csv` but we don't need to type it all, so in the text I show an abbreviation of it.

```
$ gnuplot
gnuplot> set datafile separator comma
gnuplot> plot 'world-pop.csv' using 1:3 with linespoints
```

You can see the results of this in Figure 1.4.

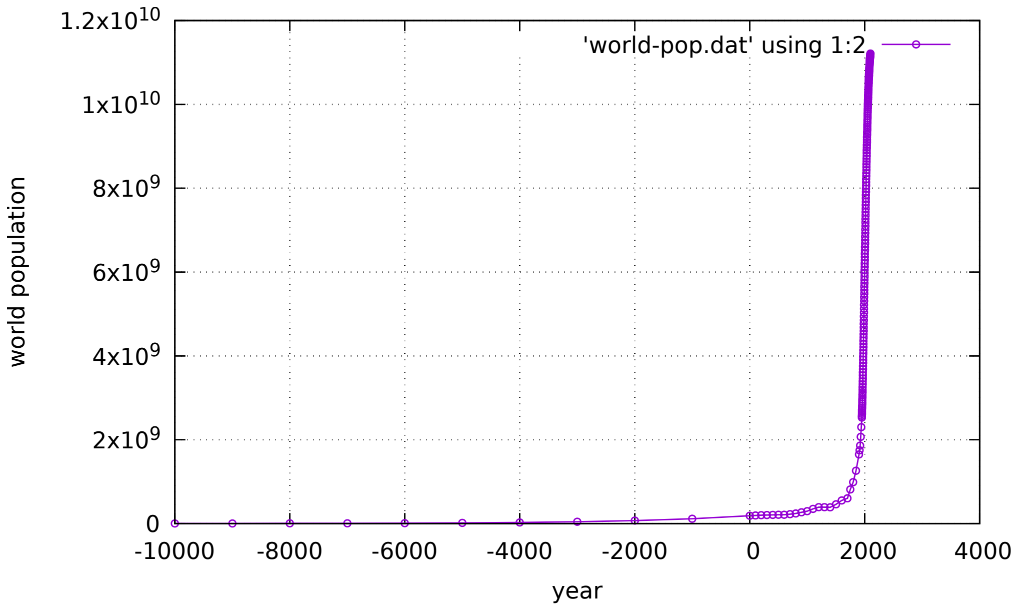

Figure 1.4: World population from 10000 BCE to 2100 CE (projected after the present). The lower plot uses a "log scale" on the y axis. This plot was generated by these instructions:

commands to generate data

```
wget --continue http://ourworldindata.org/roser/graphs/WorldPopulationAnnual12000years_interpolated_HYDEandUN/WorldPopula
sed 's/,/   /g' world-pop.csv | tr '\r' '\n' > world-pop.dat
```

gnuplot instructions

```
#set multi layout 2, 1
set grid
set xlabel 'year'
set ylabel 'world population'
plot 'world-pop.dat' using 1:2 with linespoints pt 6 ps 0.4
#set logscale y
#plot 'world-pop.dat' using 1:2 with linespoints pt 6 ps 0.4
```

And what a story we could tell from this plot if it weren't so hard to read! The main problem with this plot is that the world population in ancient times was quite small, and then it grew dramatically with various milestones in history which allowed for longer life expectancy and for the occupation of more of the world.

There are a couple of ways of trying to get more out of this plot. One is to *zoom in* to certain parts of it. For example, in Figure 1.5 we zoom in to the milennium from the founding of Rome to the fall of the western Roman empire.

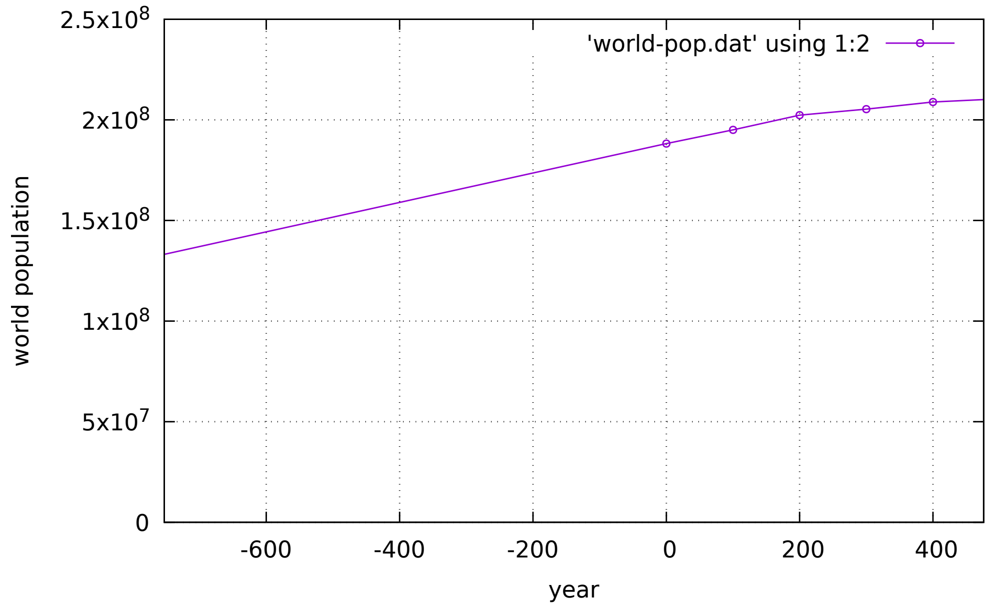

Figure 1.5: World population from 753 BCE (founding of Rome) to 476 CE (fall of the western Roman empire). This plot was generated by these instructions:

commands to generate data

```
wget --continue http://ourworldindata.org/roser/graphs/WorldPopulationAnnual12000years_interpolated_HYDEandUN/WorldPopula
sed 's/,/   /g' world-pop.csv | tr '\r' '\n' > world-pop.dat
```

gnuplot instructions

```
####CAPTION: World population from -146 BCE (Rome destroys Carthage)
set grid
set xlabel 'year'
set ylabel 'world population'
plot [-753:476] [0:] 'world-pop.dat' using 1:2 with linespoints pt 6 ps 0.4
#set logscale y
#plot 'world-pop.dat' using 1:2 with linespoints pt 6 ps 0.4
```

And in Figure 1.6 we zoom in to the 20th century.

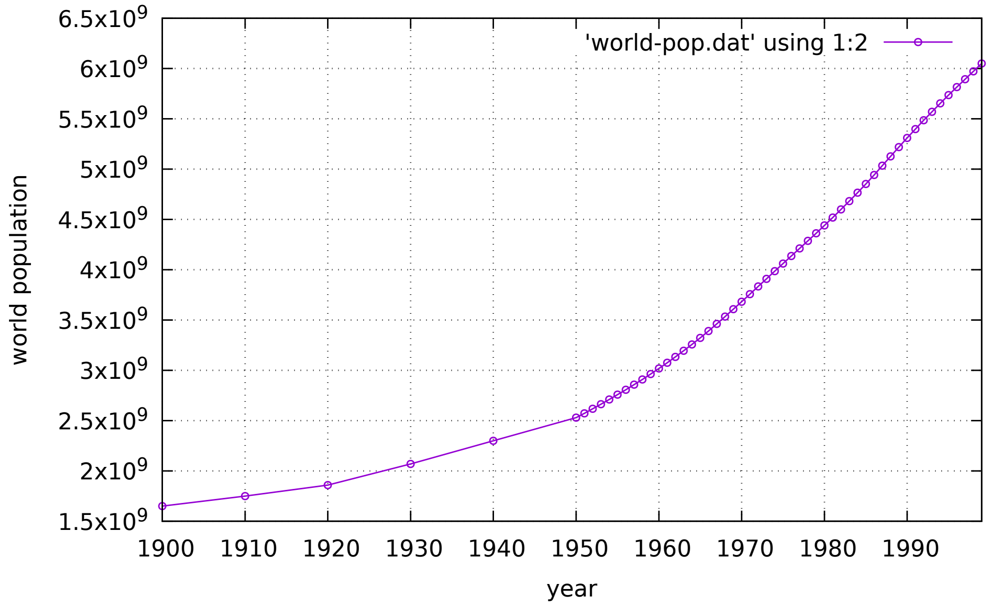

Figure 1.6: World population in the 20th century. This plot was generated by these instructions:

commands to generate data

```
wget --continue http://ourworldindata.org/roser/graphs/WorldPopulationAnnual12000years_interpolated_HYDEandUN/WorldPopula
sed 's/,/   /g' world-pop.csv | tr '\r' '\n' > world-pop.dat
```

gnuplot instructions

```
set grid
set xlabel 'year'
set ylabel 'world population'
plot [1900:1999] 'world-pop.dat' using 1:2 with linespoints pt 6 ps 0.4
#set logscale y
#plot 'world-pop.dat' using 1:2 with linespoints pt 6 ps 0.4
```

And in Figure 1.6 we zoom in to the period from year 0 to 1800 CE.

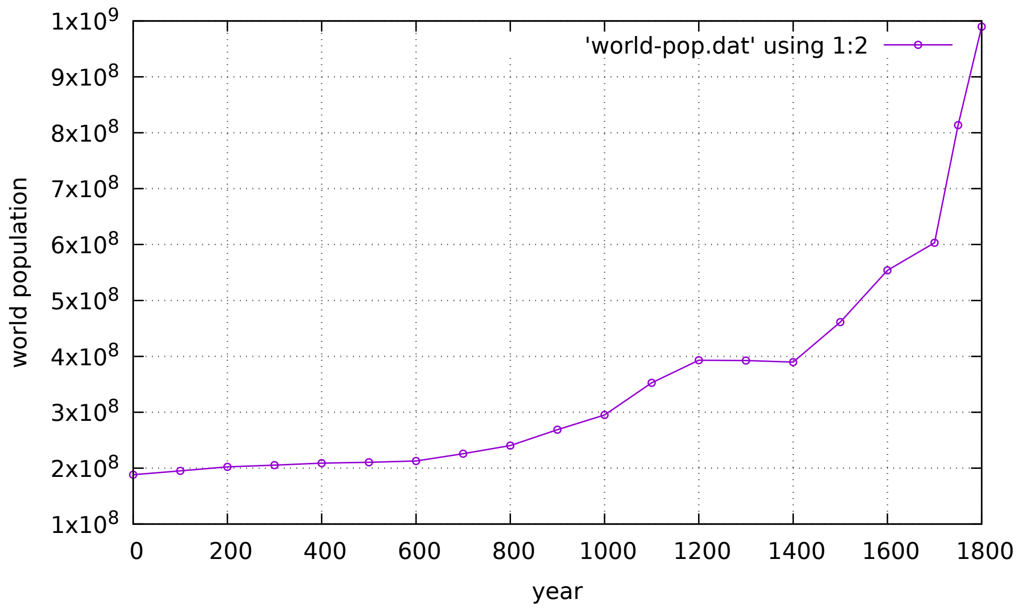

Figure 1.7: World population from year 0 to 1800 CE. This plot was generated by these instructions:

commands to generate data

```
wget --continue http://ourworldindata.org/roser/graphs/WorldPopulationAnnual12000years_interpolated_HYDEandUN/WorldPopula
sed 's/,/   /g' world-pop.csv | tr '\r' '\n' > world-pop.dat
```

gnuplot instructions

```
set grid
set xlabel 'year'
set ylabel 'world population'
plot [0:1800] 'world-pop.dat' using 1:2 with linespoints pt 6 ps 0.4
#set logscale y
#plot 'world-pop.dat' using 1:2 with linespoints pt 6 ps 0.4
```

These attempts at zooming in tell us a some interesting things:

- It is frustrating that there is so little data before 1950.
- The 0 to 1800 plot allows us to see things clearly before the population jumps up so much.
- In the 0-1800 plot we see that the world population starts growing as we approach the year 1000, after which it flattens off around the year 1300 (the period of the great plague), after which it starts pick up and never stops growing.

The other way to look at data when the y axis has too much range is to use what is called a *log scale*. Figure 1.8 shows how this can be done in `gnuplot`, and you can see that the y axis has been adjusted so that we can see some of the features in the data, especially compared to Figure 1.4.

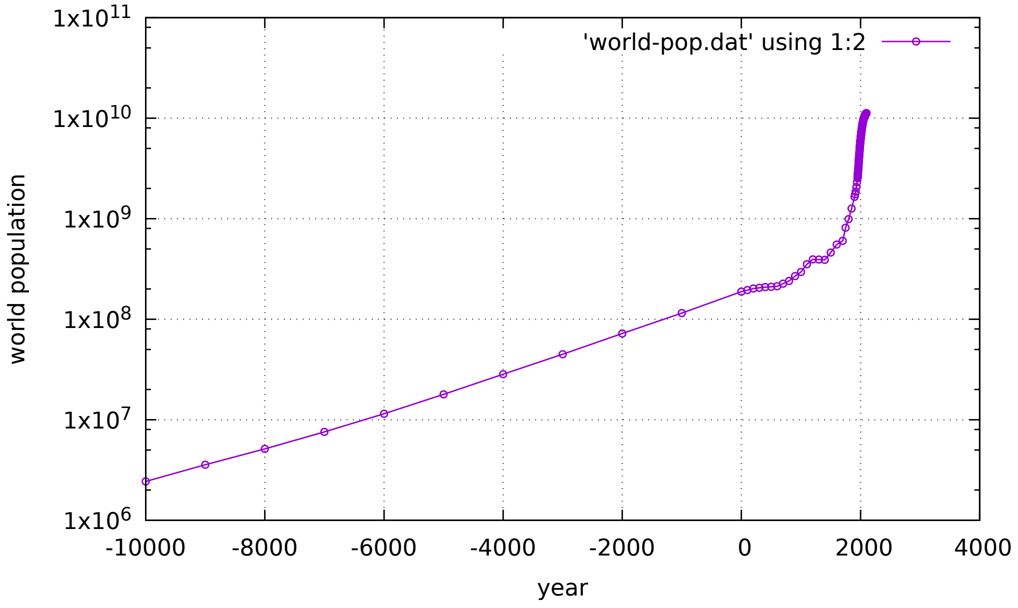

Figure 1.8: World population from 10000 BCE to 2100 CE (projected after the present). This plot uses a "log scale" on the y axis. This plot was generated by these instructions:

commands to generate data

```
wget --continue http://ourworldindata.org/roser/graphs/WorldPopulationAnnual12000years_interpolated_HYDEandUN/WorldPopula
sed 's/,/   /g' world-pop.csv | tr '\r' '\n' > world-pop.dat
```

gnuplot instructions

```
#set multi layout 2, 1
set grid
set xlabel 'year'
set ylabel 'world population'
set logscale y
plot 'world-pop.dat' using 1:2 with linespoints pt 6 ps 0.4
#set logscale y
#plot 'world-pop.dat' using 1:2 with linespoints pt 6 ps 0.4
```

1.8 Simple surface plot

So far we have looked at *line plots*. Let us now look at another type of plot: the *surface plot*, shown in Figure 1.9. This shows the function $z = e^{-(x^2+y^2)}/10$ as a height over the (x, y) position in the plane.

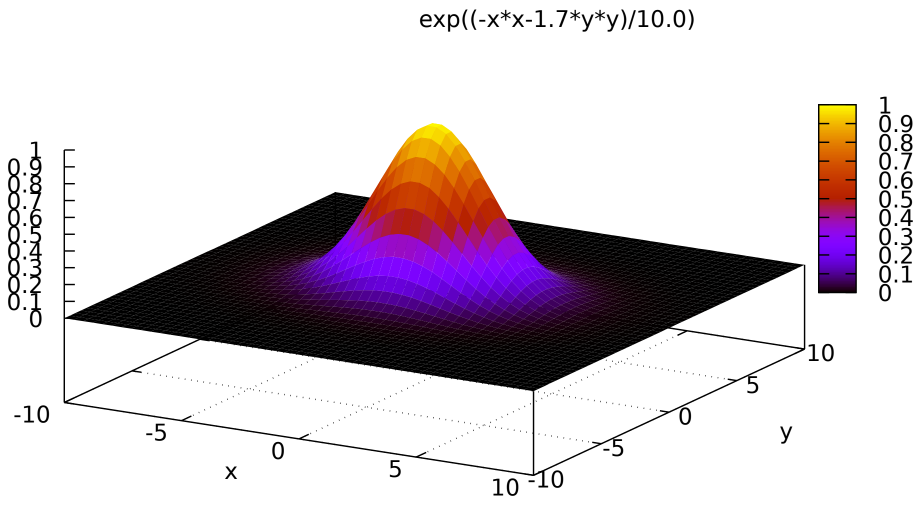

Figure 1.9: Simple surface plot. This plot was generated by these instructions:

gnuplot instructions

```
set grid
set pm3d
set xlabel 'x'
set ylabel 'y'
set samples 50
set isosamples 50
splot exp((-x*x-1.7*y*y)/10.0) with pm3d
```

Another way of showing the same information is a *heat map*. Figure 1.10 shows a similar function $z = e^{-(x^2+1.7*y^2)}/50$ where the value is represented with *color* instead of height.

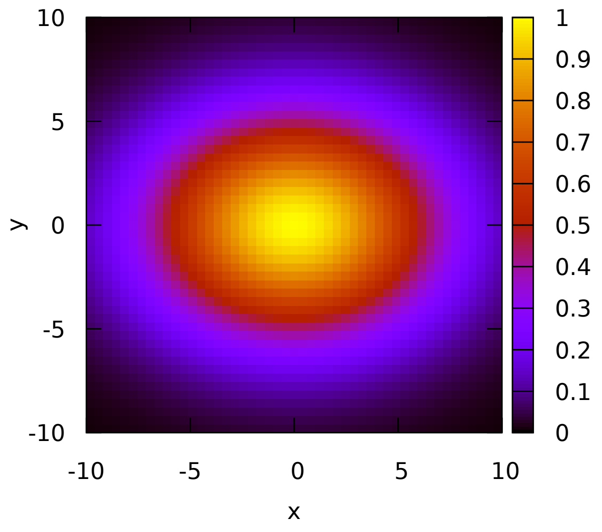

Figure 1.10: Heat map plot. This plot was generated by these instructions:

gnuplot instructions

```
set size ratio -1
set view map
set samples 50
set isosamples 50
set xlabel 'x'
set ylabel 'y'
splot exp((-x*x-1.7*y*y)/50.0) with pm3d
```

1.9 Topics we have covered

- data files
- plots
- gnuplot
- reading features from simple plots
- simple surface plots

Lesson 2

Getting and plotting temperature data

2.1 Using Python to retrieve data

The most important thing we will do in this lesson is to *write a Python program which retrieves data from the web.* This is very useful in at least these two cases:

- You want to access data that is regularly updated by some other institution. Examples of this include atmospheric, earthquake, astronomical and financial data.
- You have your own measurement device which offers up its own data using a web server.

Python has excellent libraries for accessing and parsing web data.

The example I will give here is to access the NOAA (National Oceanic and Atmospheric Administration) US Climate Research Network data. Specifically: they have weather stations in many locations around the US and they give straightforward online access to the data.

The first thing to do is find the URLs and understand how they are structured, so that we can then write our program. The base URL actually looks like this: `ftp://ftp.ncdc.noaa.gov/pub/data/uscrn/products/subhourly01/`

There are three things to notice here: *(a)* the `README.txt`, *(b)* `HEADERS.txt`, and *(c)* all the subdirectories for each year of data.

We start by exploring: if you follow the year (let us say 2014) you find a list of states and cities which have weather stations, and we will pick Las_Cruces, NM (New Mexico). The URL for Las Cruces, 2014, is:

ftp://ftp.ncdc.noaa.gov/pub/data/uscrn/products/subhourly01/2014/CRNS0101-05-2014-
Las_Cruces_20_N.txt

and in general the format is:

ftp://ftp.ncdc.noaa.gov/pub/data/uscrn/products/subhourly01/YYYY/CRNS0101-05-YYYY-
THE_CITY_20_N.txt

where YY is the year, SS is the date, and THE_CITY is the city.

The problem is: we cannot just write out the URL – we have to give the students a procedure for finding this data, so we start with a search string:

`NOAA subhourly data`

and the first match we find will then give us a link for the subhourly data sets.

Now that we have explored the layout of the data directories, we take a look at the `README.txt` and `HEADERS.txt` files. These tell us that column 9 has the air temperature, while the date and time (in UTC) are in columns 4 and 5. (Remember that in Python, as in most computer languages, arrays start at 0, so these will be positions 8, 3, 4.)

We are now ready to write a python program which downloads the 2014 data file for Las Cruces:

```python
#! /usr/bin/env python3

"""Retrieve temperatures from the NOAA USCRN weather station in Las
Cruces (actually in the mountains north of Las Cruces, as you can see
from the frequent negative temperatures). All temperatures are in
degrees Celsius.

"""

las_cruces_2014_url = \
      ('ftp://ftp.ncdc.noaa.gov/pub/data/uscrn/products'
       + '/subhourly01/2014'
       + '/CRNS0101-05-2014-NM_Las_Cruces_20_N.txt')

import urllib.request
import datetime

def main():
    time_temp_data = []
    print('## retrieving from %s' % las_cruces_2014_url)
    with urllib.request.urlopen(las_cruces_2014_url) as f:
        for line in f.readlines():
            words = line.split()
            date_yyyymmdd = words[1]
            time_hhmm = words[2]
            temp_5min_avg_str = words[8]
            ## now parse apart the date and time
            y = int(date_yyyymmdd[0:4])
            mo = int(date_yyyymmdd[4:6])
            d = int(date_yyyymmdd[6:8])
            h = int(time_hhmm[0:2])
            mi = int(time_hhmm[2:4])
            t = datetime.datetime(y, mo, d, h, mi)
            ## now find time in seconds
            epoch = datetime.datetime.utcfromtimestamp(0)
            tsec = (t -epoch).total_seconds()
            tminutes = tsec / 60.0
            ## These data files put values of -9999 for missing
            ## temperature readings. We use a simple approach and
            ## replace them with the most recent reading:
            ## last_good_temp. At the start there is no
            ## last_good_temp so we kludge it to 10.
            last_good_temp = 10
            temp_5min_avg = float(temp_5min_avg_str)
            if temp_5min_avg != -9999:
                time_temp_data.append((tminutes, temp_5min_avg))
                last_good_temp = temp_5min_avg
            else:
                time_temp_data.append((tminutes, last_good_temp))
    write_dataset(time_temp_data, 'temperatures_Las_Cruces_2014.dat')

def write_dataset(time_temp_data, fname):
    print('## writing to file %s' % fname)
    with open(fname, 'w') as f:
        f.write('## minutes_since_start hourly_average_temperature\n')
        for datum in time_temp_data:
            minutes = datum[0] -time_temp_data[0][0]
            temp = datum[1]
            f.write('%.18g  %.5g\n' % (minutes, temp))


if __name__ == '__main__':
    main()
```

Listing 2.1: First stab at temperature retrieval: `retrieve-temperature-first.py`

This program creates an empty list, then reads data from the URL, one line at a time. Each line is split into words, and words 1 and 2 (remember: 2nd and 3rd columns) are used to get the date and time, while word 8 (9th column) has the temperature in degrees Celsius.

This introduces three new Python tricks:

urllib Lets us open files on web (or ftp) servers as if they were local files. We can use the `readlines()` method as if it were a local file.

string splitting Allows us to break the line up into words.

`datetime` This library lets us turn the date and time strings into a numeric quantity (in our case minutes) which we can plot on the x axis.

Once we have our time and temperature values we add them to the list `time_temp_data`, and once we have filled that up we write it out to a file.

2.2 Plotting the temperature data

Now let us run the program, and then plot the resulting data file:

```
$ ./retrieve-temperature-first.py
## retrieving from ftp://ftp.ncdc.noaa.gov/pub/data/uscrn [....]
## writing to file temperatures_Las_Cruces.dat
$ gnuplot
gnuplot> set grid
gnuplot> set xlabel 'time (minutes since start of data)'
gnuplot> set ylabel 'temperature (Celsius)'
gnuplot> set title 'Las Cruces temperature in 2014'
gnuplot> plot 'temperatures_Las_Cruces.dat' using 1:2 with lines
gnuplot> set terminal pdf
gnuplot> set output 'temperature-first.pdf'
gnuplot> plot 'temperatures_Las_Cruces.dat' using 1:2 with lines
gnuplot> unset output
```

Listing 2.2: Retrieve and plot temperatures for 2014.

This plot, shown in Figure 2.1, tells a clear story: the temperature starts low in January 2014, gets higher in the spring and summer, then goes back down in the next fall and winter.

Within this larger 1-year cycle we also see many fluctuations that look like spikes. If we zoom in by only plotting ten thousand and eight minutes (one week) like this:

```
gnuplot> set title 'Las Cruces temperature, first week of 2014'
gnuplot> plot [0:10080] 'temperatures_Las_Cruces.dat' using 1:2 with lines
gnuplot> set terminal pdf
gnuplot> set output 'temperature-first-one-week.pdf'
gnuplot> plot [0:10080] 'temperatures_Las_Cruces.dat' using 1:2 with lines
gnuplot> unset output
```

Listing 2.3: Plot temperatures for the first week of 2014.

we get the second plot shown in Figure 2.2.

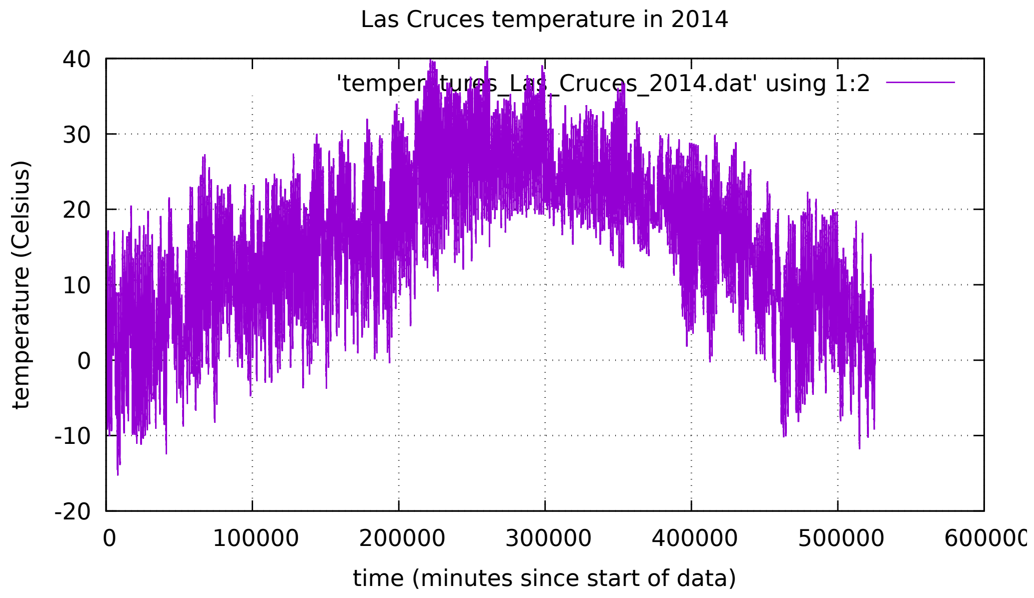

Figure 2.1: Plot generated by the following: run the commands at the shell to generate the data, then plot the data with with "`gnuplot retrieve-temperature-first-A.gp`"

commands to generate data

```
./retrieve-temperature-first.py
```

gnuplot instructions

```
## data came from ftp://ftp.ncdc.noaa.gov/pub/data/ [....]
set grid
set xlabel 'time (minutes since start of data)'
set ylabel 'temperature (Celsius)'
set title 'Las Cruces temperature in 2014'
plot 'temperatures_Las_Cruces_2014.dat' using 1:2 with lines
```

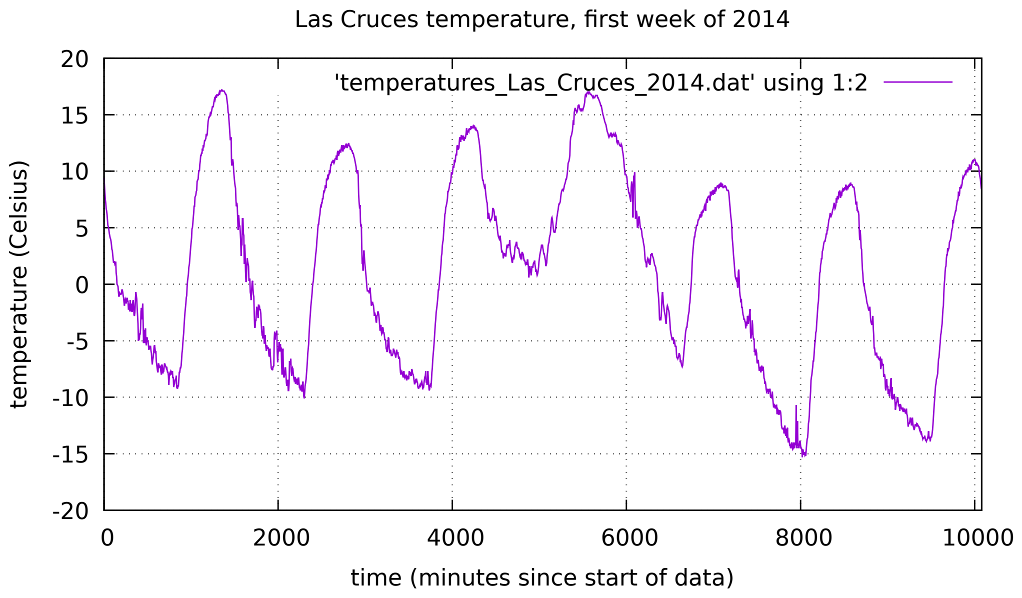

Figure 2.2: Plot generated by the following: run the commands at the shell to generate the data, then plot the data with with "`gnuplot retrieve-temperature-first-B.gp`"

commands to generate data

```
./retrieve-temperature-first.py
```

gnuplot instructions

```
## data came from ftp://ftp.ncdc.noaa.gov/pub/data/ [....]
set grid
set xlabel 'time (minutes since start of data)'
set ylabel 'temperature (Celsius)'
set title 'Las Cruces temperature, first week of 2014'
plot [0:10080] 'temperatures_Las_Cruces_2014.dat' using 1:2 with lines
```

This zoomed-in plot allows us to say that the spikes are not "noise", but rather daily fluctuations. One week has 10080 minutes, and we see seven cycles in that period of time. Within that you do see some noise, but it's clear that the main features in these plots are the yearly and daily temperature fluctuations.

2.3 Retrieving more than one year

The program in Section 2.1 only fetches one year of temperature, but it is instructive to gather a few years, since it shows us that along with the daily period (Figure 2.2), we also have an annual period (Figure 2.3).

Figure 2.2 shows a few years of plots. We will examine this superposition of two different periodic signals in greater detail in Section 4.3, using the powerful technique of fourier analaysis.

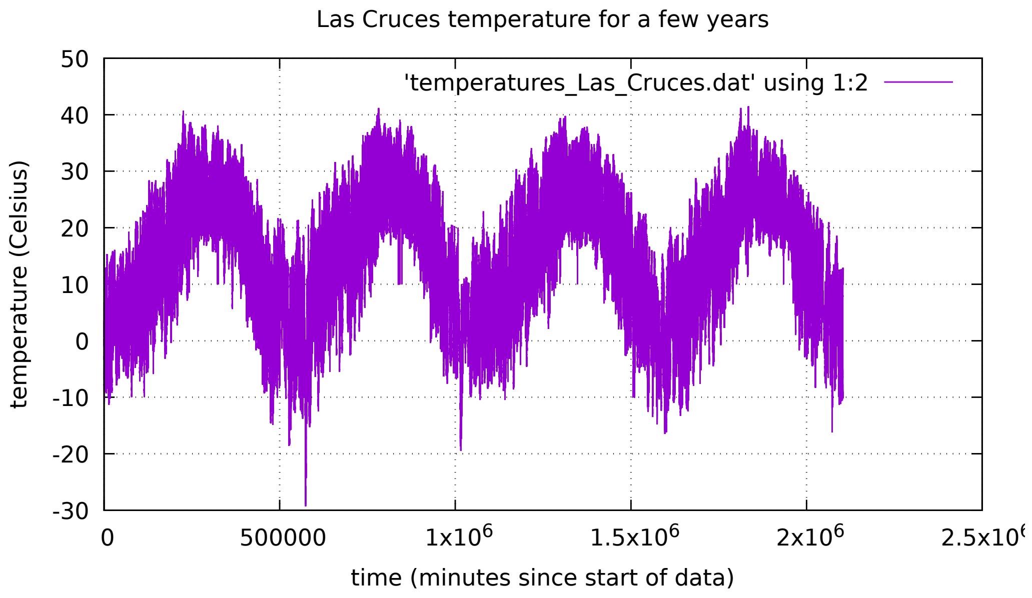

Figure 2.3: Plot generated by the following: run the commands at the shell to generate the data, then plot the data with with "`gnuplot retrieve-temperature-multi.gp`"

commands to generate data

```
./retrieve-temperature-multi.py
```

gnuplot instructions

```
## data came from ftp://ftp.ncdc.noaa.gov/pub/data/ [....]
set grid
set xlabel 'time (minutes since start of data)'
set ylabel 'temperature (Celsius)'
set title 'Las Cruces temperature for a few years'
plot 'temperatures_Las_Cruces.dat' using 1:2 with lines
```

Lesson 3

Exploring statistics

Sometimes labeled as boring, the study of statistics is quite fascinating: an area in which our intuition is often incorrect, but at the same time a discipline which is crucial for our understanding of many real-world problems.

We will use random number generators to simulate some situations in which it is very easy to produce and plot data which gives surprising but solid insights into some aspects of nature.

3.1 Poisson statistics in a time series

You have a process which generates events intermittently in time. There are at least two types of situations:

- Each event time *is related to the value of the previous event time*. Example: your next meal – you get hungry after a certain amount of time since your last meal, which affects when you next choose to eat. (Note that in a very regimented family these times might not be random at all.)

- Each event time *is **independent** of the previous event time*. Example: radioactive decay, where each particle decay time has nothing to do with the previous one.

The second type of process (event time is independent of previous event time) is called a *Poisson process*.

It is interesting to understand when a series of events comes from a poisson process and when it does not. We will now use the random number generator to simulate a poisson process, but first let us give ourselves a physical example of what might be happening.

Imagine the following. You have a house in a very unfortunate place: every day there is a 30% chance of a lightning striking that house. And every day's 30% *does not change* based on what happened yesterday or at any time in the past. To use Steven Pinker's expression (Pinker 2011), every day Zeus throws a 10-sided die, and if the number is between 1 and 3 the house will be hit.

Let us study a bit about what happens to this house. We will study it by simulating each day whether it gets hit or not, and we will collect statistics on whether it was hit.

One might wonder why lightning strikes are interesting. At this stage I point out that these sequences of random events are called a *Poisson series:* the chance of each event is unrelated to when or how the previous event happened. Other examples might include radioactive decay, earthquakes, outbreak of war, some measures in financial markes, and many others.

Let us start by reminding ourselves of how python allows us to produce random numbers. In our hacking camp (Galassi 2015) we saw this example of a few functions:

```
>>> import random
>>> random.random()
>>> random.random()
>>> random.random()
>>> random.randint(-3, 10)
>>> random.randint(-3, 10)
>>> random.randint(-3, 10)
>>> random.randint(0, 2)
>>> random.randint(0, 2)
>>> random.randint(0, 2)
>>> random.randint(0, 2)
>>> random.randint(0, 2)
```

To generate an event which looks like the result of tossing dice with a 30% outcome, and to see if the probability was right, we can use this little program:

```
#! /usr/bin/env python3
import random
def main():
    n_days = 1000
    n_hits = 0
    n_misses = 0
    for day in range(n_days):
        r = random.random()
        if r <= 0.3:           ## 30% chance
            n_hits += 1
        else:
            n_misses += 1
    hit_fraction = n_hits / n_days
    print('average daily hits: %g (%d days)' % (hit_fraction, n_days))

if __name__ == '__main__':
    main()
```

Listing 3.1: `lightning-first-stab.py` - first stab

When you run this program you should get an average lighting strikes per day that is close to 0.3:

```
$ ./lightning-first-stab.py
average daily hits: 0.285 (1000 days)
```

every time the run gives a different result, but it's always close to 0.3.

It's good to get some agility with these simple programs. Let us start by seeing how the average behaves when we simulate more or fewer days. We start by writing a function which does the simple calculation of average lightning strikes/day:

```
#! /usr/bin/env python3
import random
import math

def main():
    for n_days in (100, 1000, 10000, 100*1000, 1000*1000):
        simulate_strikes(n_days)
        simulate_strikes(n_days)
        simulate_strikes(n_days)
        simulate_strikes(n_days)
        print()

def simulate_strikes(n_days):
    n_hits = 0
    n_misses = 0
    for day in range(n_days):
        r = random.random()
        if r <= 0.03:          ## 3% chance
            n_hits += 1
        else:
            n_misses += 1
    hit_fraction = n_hits / n_days
    how_much_off = math.fabs(hit_fraction -0.03)
    print('average daily hits: %g (%d days), off by %g'
          % (hit_fraction, n_days, how_much_off))
```

```
if __name__ == '__main__':
    main()
```

Listing 3.2: `lightning-vary-n-days.py` - vary the number of days

The output of `lightning-vary-n-days.py` looks like this:

```
$ ./lightning-vary-n-days.py
average daily hits: 0.04 (100 days), off by 0.01
average daily hits: 0.05 (100 days), off by 0.02
average daily hits: 0.05 (100 days), off by 0.02
average daily hits: 0.01 (100 days), off by 0.02

average daily hits: 0.03 (1000 days), off by 0
average daily hits: 0.026 (1000 days), off by 0.004
average daily hits: 0.032 (1000 days), off by 0.002
average daily hits: 0.024 (1000 days), off by 0.006

average daily hits: 0.0303 (10000 days), off by 0.0003
average daily hits: 0.033 (10000 days), off by 0.003
average daily hits: 0.0288 (10000 days), off by 0.0012
average daily hits: 0.0281 (10000 days), off by 0.0019

average daily hits: 0.0299 (100000 days), off by 0.0001
average daily hits: 0.03062 (100000 days), off by 0.00062
average daily hits: 0.02974 (100000 days), off by 0.00026
average daily hits: 0.03062 (100000 days), off by 0.00062

average daily hits: 0.030283 (1000000 days), off by 0.000283
average daily hits: 0.030008 (1000000 days), off by 8e-06
average daily hits: 0.030252 (1000000 days), off by 0.000252
average daily hits: 0.030035 (1000000 days), off by 3.5e-05
```

The interesting thing about this output is that it shows how more runs give you an average number of lightning strikes/day that gets closer and closer to the 0.3 number.

This confirms that the the snippet of Python code which counts hits and misses:

```
        # ...
        r = random.random()
        if r <= 0.3:            ## 30% chance
            n_hits += 1
        else:
            n_misses += 1
        # ...
    hit_fraction = n_hits / n_days
```

is a valid way of simulating a random occurrence which is *uniformly distributed*. It also reminds us again that if you want good statistics you need many events: the runs with 100000 events gave an average much closer to 0.3 than the runs with 10 or 100 runs...

Let us now collect some other properties than the average number of lightning strikes. One question we might ask about these events is:

Q: what is the typical time that elapses between successive strikes?

To collect this information we modify the program and call it `lightning-time-distribution.py`

```
#! /usr/bin/env python3
import random
import math

def main():
    ## run this program with n_days = 50 when you want
    ## to eyeball the output; run it with n_days = 1000,
    ## then 10*1000, then 100*1000 when you want to make
    ## plots
    n_days = 10000
    delta_t_list = simulate_strikes(n_days)
    ## now that we have the list we print it to a file
    with open('time_diffs.dat', 'w') as f:
        for delta_t in delta_t_list:
            f.write("%d\n" % delta_t)
        print('wrote time_diffs.dat with %d delta_t values'
              % len(delta_t_list))



def simulate_strikes(n_days):
    """simulates lightning strikes for a given number of
    days, collecting information on the times between
    strikes. returns the list of delta_t values.
    """
    last_delta_t = -1
    delta_t_list = []
    prev_day_with_strike = -1
    for day in range(n_days):
        r = random.random()   # a random float between 0 and 1
        if r <= 0.3:          # 30% chance
            #print('%d: hit' % day)
            if prev_day_with_strike >= 0:
                ## we record the delta_t of this event
                last_delta_t = day -prev_day_with_strike
                delta_t_list.append(last_delta_t)
            prev_day_with_strike = day
    return delta_t_list

if __name__ == '__main__':
    main()
```

Listing 3.3: `lightning-time-distribution.py` - study the time spacing between strikes

This program outputs a list of time intervals, Δt. Let us get an idea of what these look like by making a scatter plot of the Δt values:

```
$ gnuplot
gnuplot> plot 'time_diffs.dat' using 1 with points pt 4 ps 0.3
```

This does not look like much: just some 30 points on the screen, somewhat randomly laid out. There are not enough points yet to notice a clear pattern, so change the number `n_days` to be a very large number, like 10000, and see what you get. We can re-run the program:

```
$ ./lighting-time-distribution.py 10000
$ gnuplot
gnuplot> plot 'time_diffs.dat' using 1 with points pt 4 ps 0.3 title 'time between \
    lightning strikes'
```

and see the result in Figure 3.1.

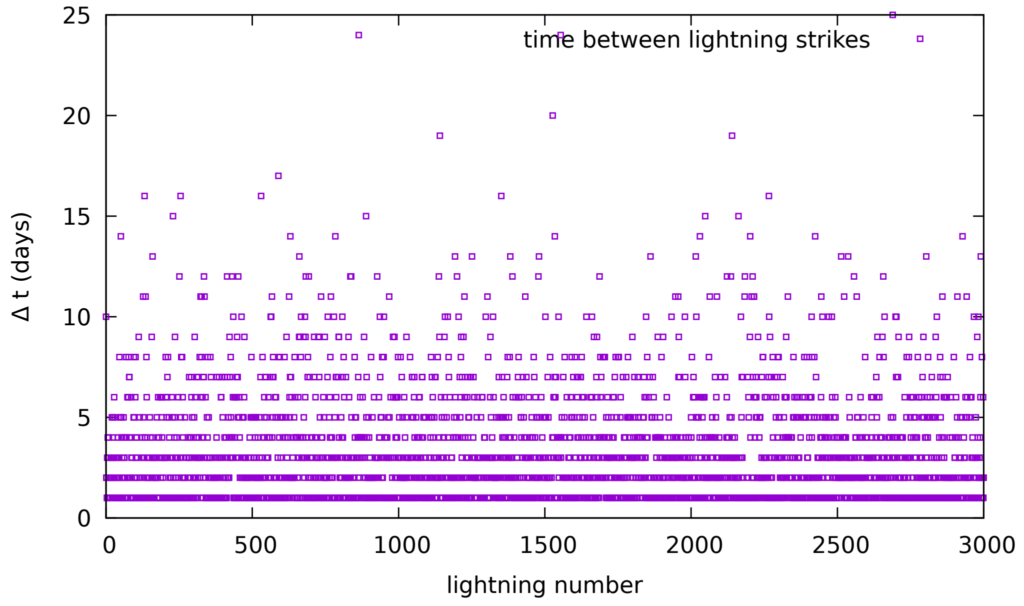

Figure 3.1: Plot generated by the following: run the commands at the shell to generate the data, then plot the data with with "`gnuplot lightning-scatter.gp`"

commands to generate data

```
./lightning-time-distribution.py
```

gnuplot instructions

```
set xlabel 'lightning number'
set ylabel '{/Symbol D} t (days)'
plot 'time_diffs.dat' using 1 with points pt 4 \
   ps 0.3 title 'time between lightning strikes'
```

Here is how I would read the plot to a class of students:

> On the x axis you just see the sequence of strikes (some 3000 of them). On the y axis you see how many days had passed since the previous strike. What is really interesting (and you probably did not guess it beforehand) is that there are many more points down below where Δt is small, and very few at high values of Δt. In particular, *there was never a period of more than 30 days between strikes.*

I would then jump up and down, exclaiming "you see how a single plot command can give you so much insight?"

To get even more insight let us show what a random plot would have looked like. The program `random-uniform.py` puts out a list of uniform random numbers:

```python
#! /usr/bin/env python3

import random

def main():
    for i in range(3000):
        print(random.randint(1, 23))

if __name__ == '__main__':
    main()
```

Listing 3.4: `random-uniform.py` – generate a list of uniform random numbers

The purely random numbers can be seen in Figure 3.2, and now we can really jump up and down yelling about insight: the scatter plot of random numbers had no structure, whereas the scatter plot of time between lightning strikes had a clear structure.

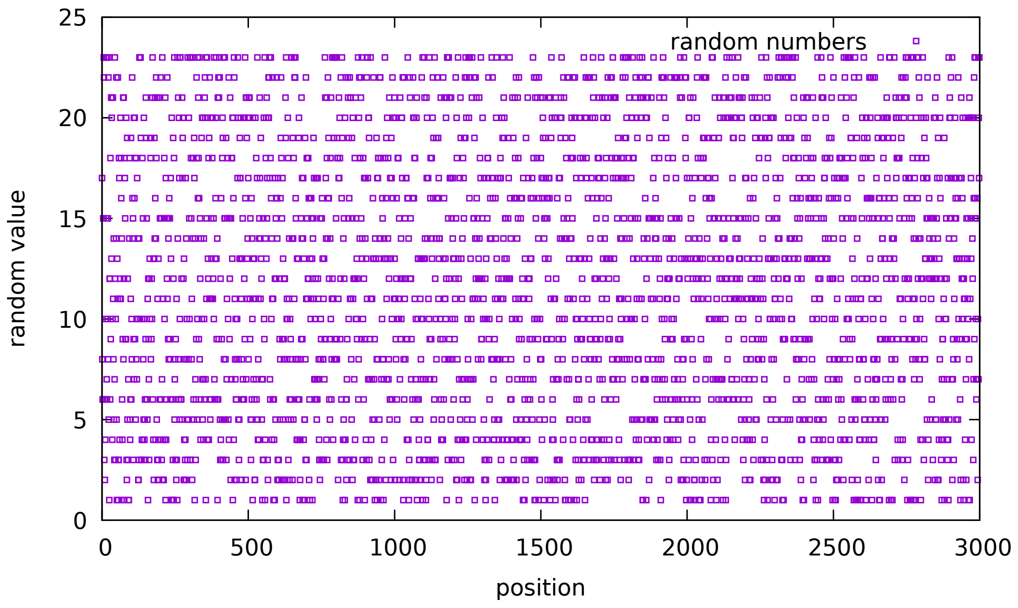

Figure 3.2: Plot generated by the following: run the commands at the shell to generate the data, then plot the data with with "`gnuplot random-scatter.gp`"

commands to generate data

```
./random-uniform.py > random_uniform.dat
```

gnuplot instructions

```
set xlabel 'position'
set ylabel 'random value'
plot 'random_uniform.dat' using 1 with points pt 4 \
    ps 0.3 title 'random numbers'
```

Then a sobering note: this "scatter plot" was very good for probing the data for a quick bit of insight, but it does not tell us anything quantitative about the time between strikes. To do this we need to plot *histograms* of the data.

3.2 Histograms of quantities

A lot of plots you are used to seeing in popular media are *histogram* plots. These plots don't show the measured quantities directly: they show how many times certain values come up.

For example, when you look at a plot of the weight of a group of people you see the typical bell curve, and on the x axis you have *weight ranges*, while on the y axis you have *how many people are in that weight range*.

The data is not naturally measured in this way, so we write a bit of code to change it to that format.

Let us do this with the file `time_diffs.dat` which was written out by our `lightning-time-distribution.py`. It contains a single column of Δt values (measured in days) that look like this:

```
1
4
12
3
2
2
5
6
```

```
1
1
8
2
## ... many more lines ...
```

What we want to do is count how many times each duration appears in the file, this will be the histogram. A python program to do so might be:

```
#! /usr/bin/env python3

"""Takes a file with a single column of integers and makes a histogram
of how frequently those integers occur in the file."""

import sys

def main():
    fname = sys.argv[1]
    histogram = []
    with open(fname, 'r') as f:
        lines = f.readlines()
        for line in lines:
            delta_t = int(line)
            ## in case this delta_t is bigger than any seen so far
            while delta_t >= len(histogram):
                histogram.append(0)
            histogram[delta_t] += 1

    hist_out_fname = sys.argv[1] + '.hist'
    with open(hist_out_fname, 'w') as f:
        for value in range(len(histogram)):
            f.write('%d   %d\n' % (value, histogram[value]))
    print('wrote histogram to %s' % hist_out_fname)

if __name__ == '__main__':
    main()
```

Listing 3.5: `int-histogram-maker.py` - make a histogram from a file of δt values

You can run it with:

```
$ ./int-histogram-maker.py time_diffs.dat
$ gnuplot
gnuplot> set grid
gnuplot> set xlabel '{/Symbol D} t (days)'
gnuplot> set ylabel 'frequency of that interval'
gnuplot> plot 'time_diffs.dat.hist' using 1:2 with linespoints
```

We can do better: plotting programs have special ways of plotting histograms – try this one:

```
gnuplot> plot 'time_diffs.dat.hist' using 1:2 with boxes
```

The result is shown in Figure 3.3

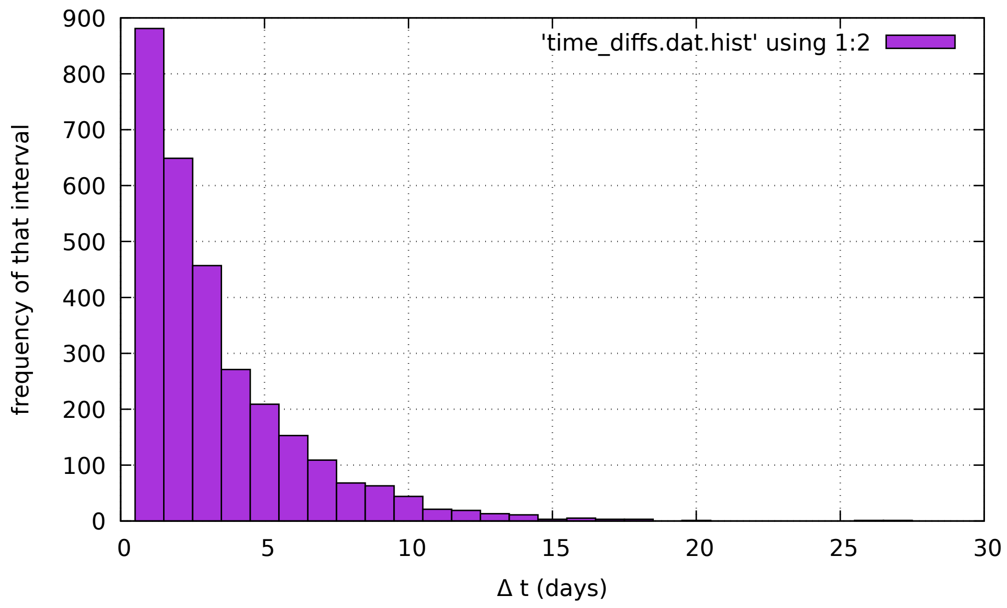

Figure 3.3: Plot generated by the following: run the commands at the shell to generate the data, then plot the data with with "`gnuplot lightning-hist.gp`"

commands to generate data

```
./lightning-time-distribution.py
./int-histogram-maker.py time_diffs.dat
```

gnuplot instructions

```
set grid
set xlabel '{/Symbol D} t (days)'
set ylabel 'frequency of that interval'
set style data histogram
set style fill solid 0.8 border -1
plot [0:] [0:] 'time_diffs.dat.hist' using 1:2 with boxes
```

The story in this plot (Figure 3.3) is easy to tell: there are many more lightning strikes that are closely spaced than that are far apart.

For completeness let us look at what happens when we take the random values shown in the scatter plot and look at this histogram of those? The result (a flat histogram) is shown in Figure3.4.

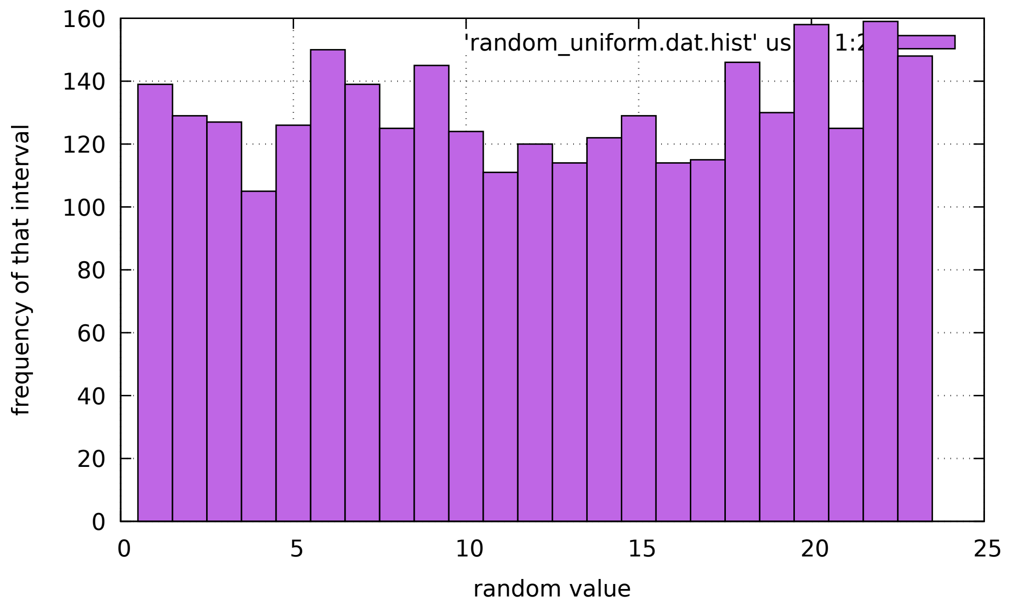

Figure 3.4: Plot generated by the following: run the commands at the shell to generate the data, then plot the data with with "`gnuplot random-hist.gp`"

commands to generate data

```
./random-uniform.py > random_uniform.dat
./int-histogram-maker.py random_uniform.dat
```

gnuplot instructions

```
set grid
set xlabel 'random value'
set ylabel 'frequency of that interval'
set style data histogram
set style fill solid 0.6 border -1
plot [0:] [0:] 'random_uniform.dat.hist' using 1:2 with boxes
```

3.3 Random spatial distribution

We have talked about processes which give events distributed *randomly in time:* events happen at random times. Let us now look at processes that generate points distributed *randomly in space:* (x, y) coordinates are spewed out by our process. An example might be where the grains of sand land when you drop a handful onto the ground.

We can write a program to generate random (x, y) points between 0 and 100. The program `random-spatial.py` generates a series of such points, each completely independent of the previous one.

```python
#! /usr/bin/env python3

"""
Print a bunch of (x, y) points.
"""

import random

def main():
    for i in range(3000):
        x = random.randint(0, 100)
        y = random.randint(0, 100)
        print('%d   %d' % (x, y))

if __name__ == '__main__':
    main()
```

Listing 3.6: `random-spatial.py` – generates random points in space.

The results of running this program are shown in 3.5. You can see *features* in the data, even though it was randomly generated: filaments, clustering, voids…[1]

[1]Note that the clustering is an artifact of the random generation of points; it is not due to a physical effect that clusters the points together.

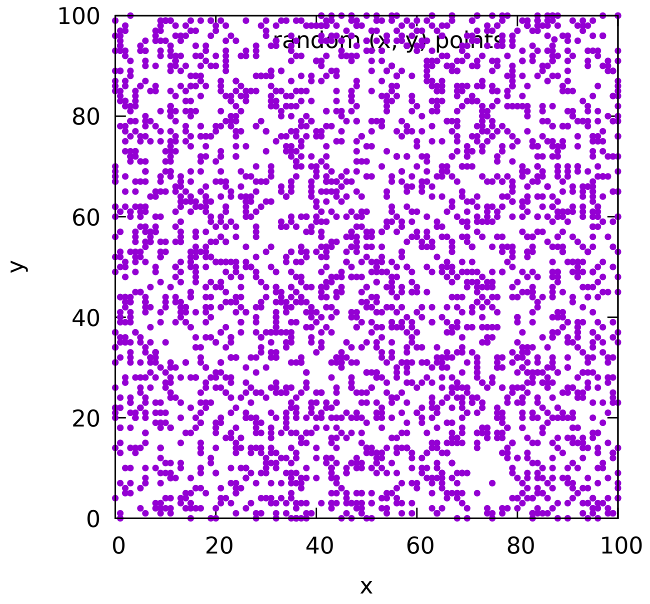

Figure 3.5: Random (x, y) points. You should be able to see some structure: occasional filaments, clustering, and empty spaces. This plot was generated by these instructions:

commands to generate data

```
./random-spatial.py > random_spatial.dat
```

gnuplot instructions

```
set size ratio -1
set xlabel 'x'
set ylabel 'y'
plot 'random_spatial.dat' using 1:2 with points pt 7 \
    ps 0.3 title 'random (x, y) points'
```

A possible comment: people who spend a lot of time looking at randomly generated data probably don't easily believe in conspiracy theories.

We can then do something analogous to what we did for random events in time: plot the distribution of distances between (x, y) points. The programs `xy-to-distances.py` and `int-histogram-maker` allow us to do so, and the results are in Figure 3.6. Note that you will not get as much insight out of these spatial histograms as you did in Figure 3.3, since a big factor in the distribution of spacial distances is the small size of the x-y plane we used.

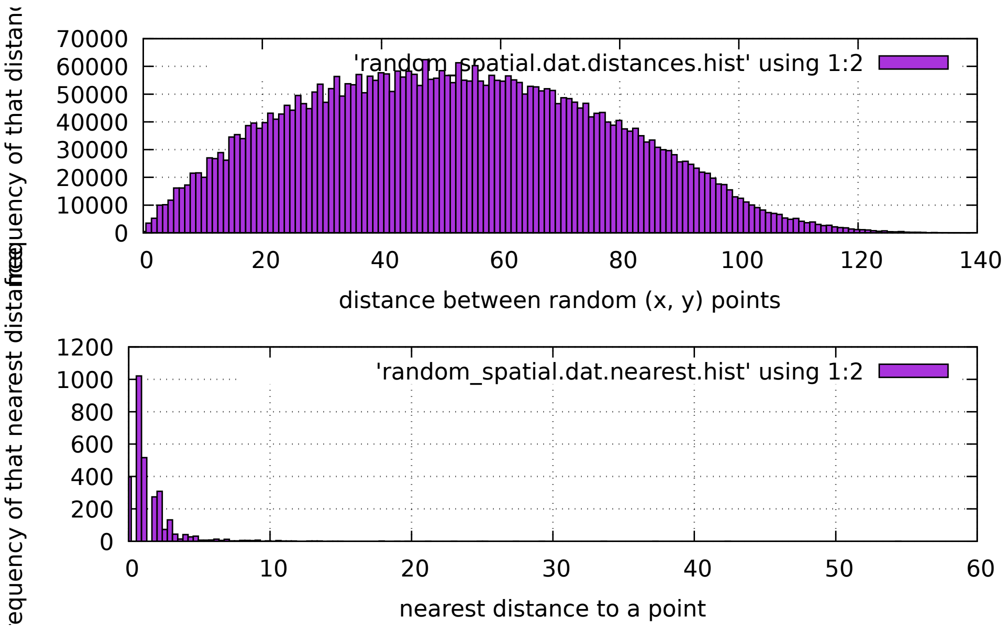

Figure 3.6: Plot generated by the following: run the commands at the shell to generate the data, then plot the data with with "`gnuplot spatial-hist.gp`"

commands to generate data

```
./random-spatial.py > random_spatial.dat
./xy-to-distances.py random_spatial.dat
./float-histogram-maker.py random_spatial.dat.distances
./float-histogram-maker.py random_spatial.dat.nearest
```

gnuplot instructions

```
set grid
set multi layout 2,1
set style data histogram
set style fill solid 0.8 border -1
set xlabel 'distance between random (x, y) points'
set ylabel 'frequency of that distance'
plot [0:] 'random_spatial.dat.distances.hist' using 1:2 with boxes
set xlabel 'nearest distance to a point'
set ylabel 'frequency of that nearest distance'
plot [0:] 'random_spatial.dat.nearest.hist' using 1:2 with boxes
```

3.4 What have we learned

In this section we have learned that:

- A histogram shows you *how often* a value comes up (the frequency of certain values).
- We can write simple Python programs which take data and make histograms of the frequency of those values.
- Plotting programs can be used to see the histograms.

Lesson 4

Looking deeply at a curve

There is often more to a plot than immediately meets the eye, and in the life of a scientist one of the great joys is to glean more information than what is on the surface.

Here we will show one of my favorite tools for extracting further information from data: the *Fourier transform*, also referred to as the *Fourier spectrum*.

It is unlikely that you will be able to fully explain Fourier Transforms to kids, but here is an approach:

> Now we will now talk about a tool that seems almost magical which lets us finding surprising information in data. This involves some higher math, so I will ask you to accept some of it on faith for now. First of all, the tool is called the *Fourier Transform* – repeat that after me ...
>
> This tool tells us that any signal at all (and here you can flash to the temperature data) can be thought of as a sum of sin waves with different frequencies. Crazy, right? But let's see if it's true.

4.1 Fourier analysis: the square wave

Let us start gnuplot and plot a sin wave, then look at some sin waves with higher frequencies. Then add them together and see what you get:

```
$ gnuplot
gnuplot> set samples 1000
gnuplot> plot [] [-1.2:1.2] sgn(sin(x))
gnuplot> replot sin(x)
gnuplot> replot (1.0/3)*sin(3*x)
gnuplot> replot (1.0/5)*sin(5*x)
gnuplot> replot sin(x) + (1.0/3)*sin(3*x) + (1.0/5)*sin(5*x)
## now look at the summed-up plot by itself:
gnuplot> plot sin(x) + (1.0/3)*sin(3*x) + (1.0/5)*sin(5*x) t '5x'
```

Listing 4.1: First look at sin waves

You should start seeing that you go from a sin wave to something that looks a bit more like a square wave. Figure 4.1 shows what the individual sin waves look like and shows how you can add up to 13 of them and get something that starts to look quite square instead of wavy.

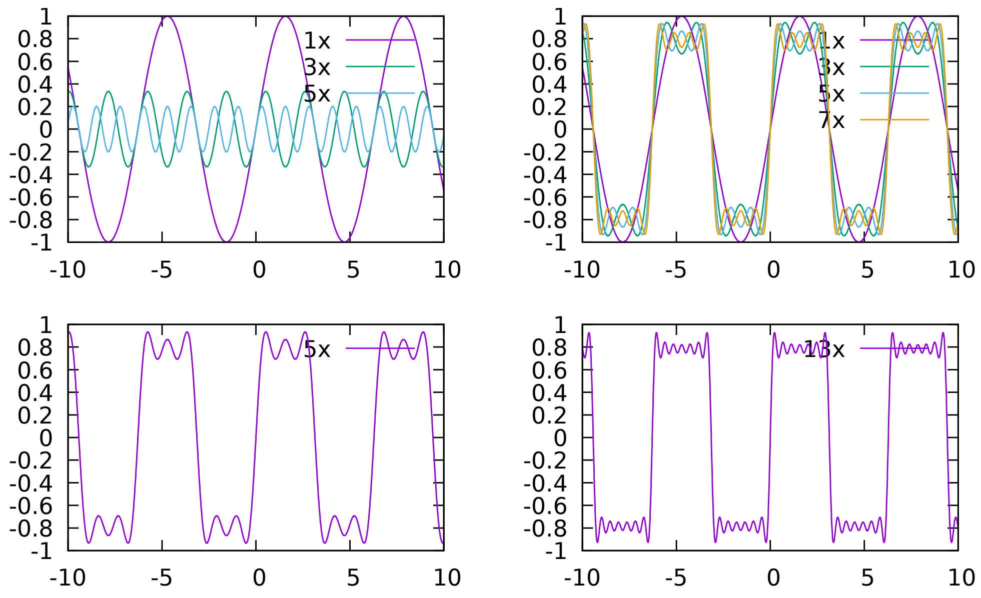

Figure 4.1: Plot generated by running gnuplot with “`gnuplot square-wave.gp`”

gnuplot instructions

```
set multi layout 2,2
set samples 3000
plot sin(x) t '1x', (1.0/3)*sin(3*x) t '3x', (1.0/5)*sin(5*x) t '5x'
plot sin(x) t '1x', \
     sin(x) + (1.0/3)*sin(3*x) t '3x', \
     sin(x) + (1.0/3)*sin(3*x) + (1.0/5)*sin(5*x) t '5x', \
     sin(x) + (1.0/3)*sin(3*x) + (1.0/5)*sin(5*x) \
            + (1.0/7)*sin(7*x) t '7x'
plot sin(x) + (1.0/3)*sin(3*x) + (1.0/5)*sin(5*x) t '5x'
plot sin(x) + (1.0/3)*sin(3*x) + (1.0/5)*sin(5*x) \
            + (1.0/7)*sin(7*x) + (1.0/9)*sin(9*x) \
            + (1.0/11)*sin(11*x) + (1.0/13)*sin(13*x) t '13x'
```

The mathematics behind the Fourier Transform are beautiful but more advanced than this course, so we will just stick with having seen the main idea: "You can take any signal and represent it as a sum of sin waves with different frequencies and amplitudes."

The square wave is not particularly realistic, so let us look at some real signals. We will start by looking at white noise, then a tuning fork, then single notes on musical instruments, then at a more complex music clip.

4.2 Fourier analysis: sound and music

When we mentioned waves with different frequencies you might have thought of sound, and you would have been right. Fourier analysis is a good tool for understanding what makes up sound waves.

Let us take a tour through a series of signals, and we will look at their fourier transforms.

4.2.1 Tuning fork

A tuning fork puts out a very pure single sin wave at a "middle A" note, also known as A_4 or concert pitch. A_4 has a frequency of 440 Hz.

We can download a stock mp3 file of the sound of a tuning fork, then we can use standard command line utilities to convert it to a text file. Once we have it as a text file we can do the following:

- Use our standard plotting techniques to see that it looks like a very clean sin wave (top plot in Figure 4.2).

- Write a program which uses the powerful Python scientific libraries to calculate the Fourier transform of the tuning fork signal.
- Look at the Fourier transform, hoping to see that a clean sin wave will appear as a single spike, indicating that there is only one sin wave in the signal.

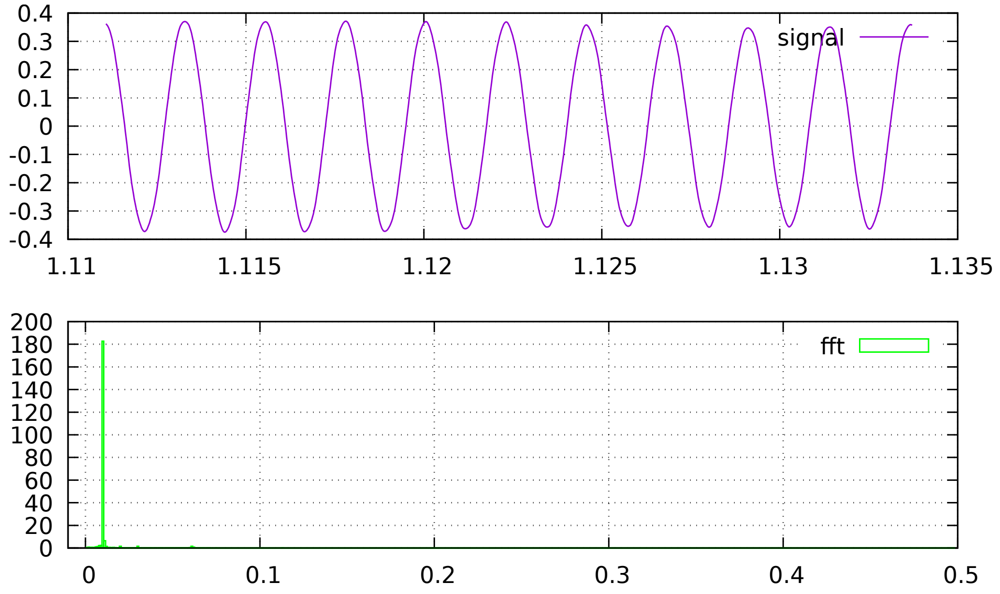

Figure 4.2: Plot generated by the following: run the commands at the shell to generate the data, then plot the data with with "`gnuplot tuningfork.gp`"

commands to generate data

```
wget --continue http://www.vibrationdata.com/tuningfork440.mp3
ffmpeg -n -i tuningfork440.mp3 tuningfork440.aiff
sox tuningfork440.aiff -t dat tuningfork440.dat
head -50000 tuningfork440.dat | tail -1000 > tuningfork-small-sample.dat
./simple-fft.py tuningfork-small-sample.dat 1 tuningfork-small-sample-fft.dat
```

gnuplot instructions

```
set multi layout 2, 1
set grid
plot 'tuningfork-small-sample.dat' using 1:2 with lines t 'signal'
plot [-0.01:] 'tuningfork-small-sample-fft.dat' using 1:3 \
     with boxes lt rgb "green" t 'fft'
```

Having looked at the plots in Figure 4.2 let us write the python program `simple-fft.py` which reads in the signal and writes out the Fourier transform:

```
#! /usr/bin/env python3

import sys
import math
import numpy as np
import scipy.signal as signal
import scipy.fftpack as fftpack

def main():
    if len(sys.argv) != 4:
        print('error: must give three arguments:')
        print('      input filename, signal column (starts at 0),')
        print('      and output filename')
        sys.exit(1)
    fin = sys.argv[1]
    signal_column = int(sys.argv[2])
    fout = sys.argv[3]
    dataset = read_dataset(fin, signal_column)
    sig_fft = find_fft(dataset)
    write_fft(sig_fft, fout)

def read_dataset(fin, signal_column):
    dataset = []
    with open(fin, 'r') as f:
        for line in f.readlines():
            if not line[0].isdigit() and not line[0].isspace():
                continue
            words = line.split()
            (t, signal) = (words[0], words[signal_column])
            dataset.append((t, signal))
    return dataset


    write_fft(sig_fft, n_days, 'fft_%s.dat' % the_city)

def find_fft(dataset):
    (time_list, signal_list) = zip(*dataset)
    times = np.array(time_list)
    signal = np.array(signal_list)
    signal_fft = fftpack.rfft(signal)

    return signal_fft

def write_fft(sig_fft, fname):
    n_samples = len(sig_fft)
    print('## writing fft to file %s' % fname)
    print('type of fft:', type(sig_fft))
    with open(fname, 'w') as f:
        f.write('## fft\n')
        f.write('## frequency frequnce-days-per-cycle fft_amplitude\n')
        fft_pts = int(len(sig_fft)/2 -1)
        for i in range(1, fft_pts):
            real_part = sig_fft[2*i-1]
            imag_part = sig_fft[2*i]
            freq = i/n_samples
            days_per_cycle = 1.0/freq
            fft_amplitude = math.sqrt(real_part**2 + imag_part**2)
            f.write('%g   %g    %g\n' % (freq, days_per_cycle, fft_amplitude))


if __name__ == '__main__':
    main()
```

Listing 4.2: Reads a simple column of signal data: `simple-fft.py`

Having written this program we can then follow the steps in Figure 4.2 to generate those plots ourselves. Note that the steps include a simple scripting trick to pick out just a small part of the tuning fork signal. That's because *(a)* these signals have more than 40000 samples/second which is big and unnecessary, and *(b)* data at the very start of the file can have artifacts.

What we need to do is skip a lot of lines at the start, and then pick out about 1000 lines of data. A simple use of UNIX shell pipelines can do this:

```
$ head -50000 tuningfork440.dat | tail -1000 > tuningfork-small-sample.dat
```

We will use `simple-fft.py` in the next few examples as well, getting a surprise amount of mileage from one program.

4.2.2 White noise

Now let us download an example of *white noise*. This is a random wave with no discernible pattern.

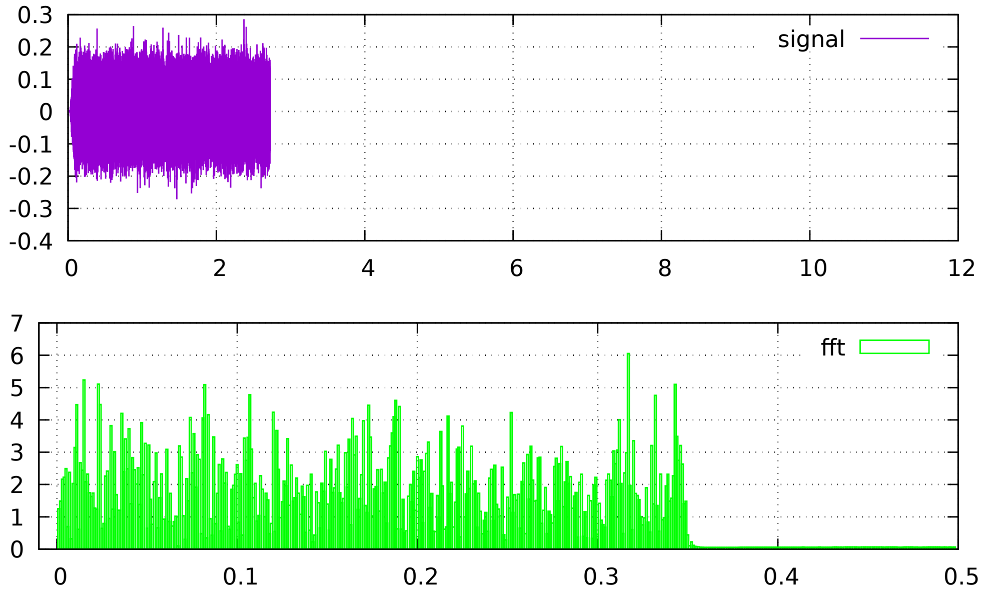

Figure 4.3: Plot generated by the following: run the commands at the shell to generate the data, then plot the data with with "`gnuplot white-noise.gp`"

commands to generate data

```
wget -q --continue http://www.vibrationdata.com/white1.mp3 -O white-noise.mp3
ffmpeg -n -i white-noise.mp3 white-noise.aiff
sox -q white-noise.aiff -t dat white-noise.dat
head -7000 white-noise.dat | tail -1000 > white-noise-small-sample.dat
./simple-fft.py white-noise-small-sample.dat 1 white-noise-fft.dat
```

gnuplot instructions

```
set multi layout 2, 1
set grid
plot 'white-noise.dat' using 1:2 with lines t 'signal'
plot [-0.01:] 'white-noise-fft.dat' using 1:3 \
     with boxes lt rgb "green" t 'fft'
```

Go ahead and follow the instructions in Figure 4.3. To the ear the signal sounds like a hissing sound, as you can see if you run

```
vlc white-noise.mp3
```

Remember this simple way of playing audio clips.

You could say that the Fourier spectrum for white noise is the opposite of that for the pure tuning fork signal: instead of a single spike you have random-looking spikes all over the spectrum.

4.2.3 Violin playing single "A" note

Now let us look at some musical notes from real instruments. Each note corresponds to a certain frequency (Backus 1977). Figure 4.4 shows a much more complex signal than the tuning fork.

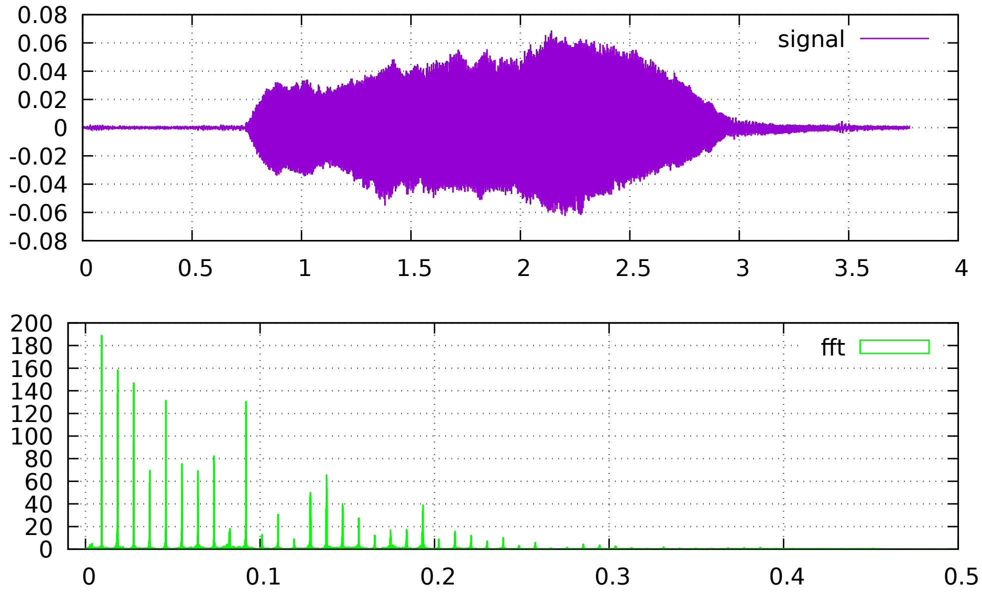

Figure 4.4: Plot generated by the following: run the commands at the shell to generate the data, then plot the data with with "`gnuplot violin-A-440.gp`"

commands to generate data

```
wget --continue http://freesound.org/data/previews/153/153587_2626346-lq.mp3 -O violin-A-440.mp3
ffmpeg -n -i violin-A-440.mp3 violin-A-440.aiff
sox violin-A-440.aiff -t dat violin-A-440.dat
./simple-fft.py violin-A-440.dat 1 violin-A-440-fft.dat
```

gnuplot instructions

```
set multi layout 2, 1
set grid
plot 'violin-A-440.dat' using 1:2 with lines t 'signal'
plot [-0.01:] 'violin-A-440-fft.dat' using 1:3 \
     with boxes lt rgb "green" t 'fft'
```

The signal in Figure 4.4 has some structure to it. The Fourier transform has several peaks: the strongest peak (for a A_4 note) will be at the *fundamental frequency* of 440 Hz, but there are many other peaks. The pattern of the peaks characterizes the sound of that specific violin (or guitar or other instrument).

4.2.4 Violin playing single "F" note

This is a repeat of Section 4.2.3 but with a different note on the violin: F instead of A. Figure 4.5 is hard to distinguish from Figure 4.4 since the character of the instrument is the same. At this point we have not written code to plot the actual frequencies on the x axis, so we cannot spot that the A_4 note has its highest peak at 440Hz, while the F_5 is at 698.45Hz (Backus 1977).

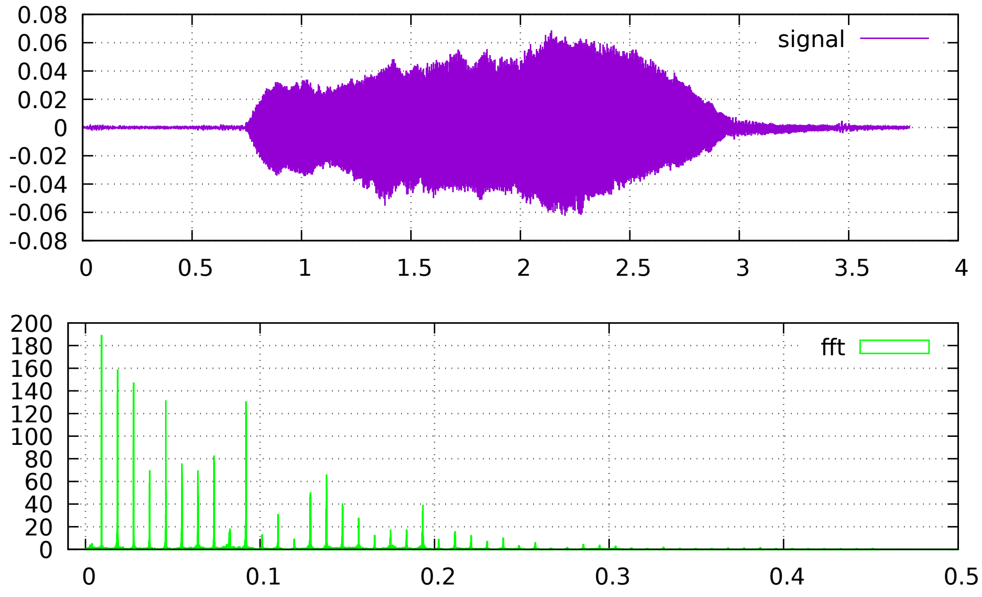

Figure 4.5: Plot generated by the following: run the commands at the shell to generate the data, then plot the data with with "`gnuplot violin-F.gp`"

commands to generate data

```
wget --continue http://freesound.org/data/previews/153/153595_2626346-lq.mp3 -O violin-F.mp3
ffmpeg -n -i violin-F.mp3 violin-F.aiff
sox violin-F.aiff -t dat violin-F.dat
./simple-fft.py violin-F.dat 1 violin-F-fft.dat
```

gnuplot instructions

```
set multi layout 2, 1
set grid
plot 'violin-F.dat' using 1:2 with lines t 'signal'
plot [-0.01:] 'violin-F-fft.dat' using 1:3 \
     with boxes lt rgb "green" t 'fft'
```

FIXME the web site with these violin clips has something weird where it always downloads the same file unless you sign in, so at this time Figure 4.5 is incorrect and probably the same as Figure 4.4. Plan: record guitar clips for the same notes and save them somewhere online.

4.2.5 A more complex music clip

The Pachelbel Canon is a well-known piece of baroque classical music which starts with a single note that is held for a while. Figure 4.6 shows the signal and its Fourier transform, where you can identify a dominant peak.

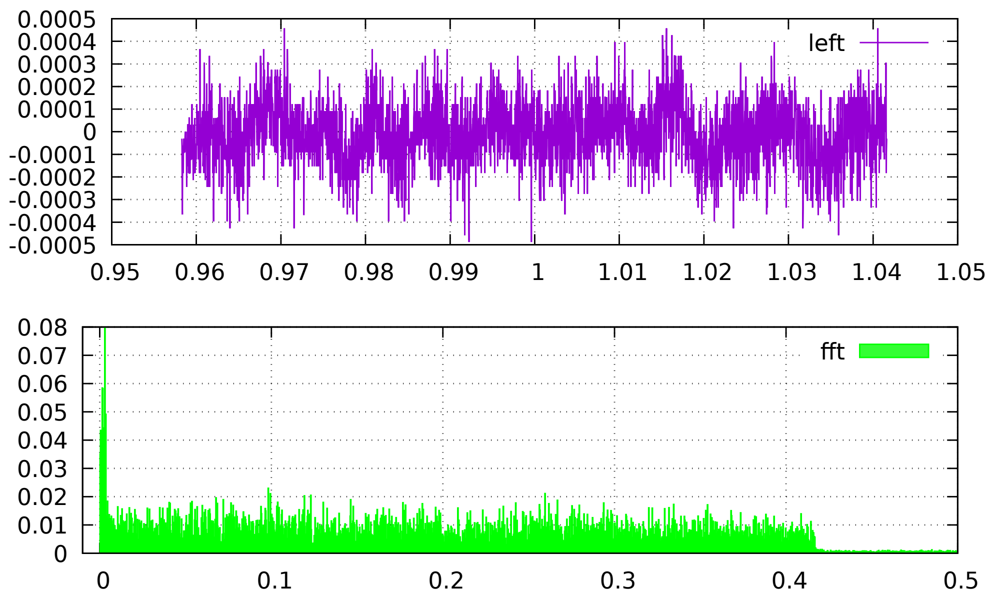

Figure 4.6: Plot generated by the following: run the commands at the shell to generate the data, then plot the data with with "`gnuplot canon.gp`"

commands to generate data

```
wget -q --continue http://he3.magnatune.com/music/Voices%20of%20Music/Concerto%20Barocco/21-Pachelbel%20Canon%20In%20D%20
ffmpeg -n -i 'Canon.mp3' Canon.aiff
sox -q Canon.aiff -t dat Canon.dat
head -50000 Canon.dat | tail -4000 > canon-small-sample.dat
./simple-fft.py canon-small-sample.dat 1 canon-small-sample-fft.dat
```

gnuplot instructions

```
set multi layout 2, 1
set grid
set style fill solid 0.8
plot 'canon-small-sample.dat' using 1:2 with lines t 'left'
plot [-0.01:] 'canon-small-sample-fft.dat' using 1:3 \
     with boxes lt rgb "green" t 'fft'
```

But what happens if we have music that is not a single note? In Figure 4.7 we will look at a clip of a choir singing *Gloria in Excelsis Deo*. The clip (which you can play with `vlc gloria.ogg` after downloading it) starts in a place where many voices are singing in harmony, so there is no single note to be picked out.

We expect to see several different peaks in the Fourier spectrum, and that is what you see in the second plot of Figure 4.7.

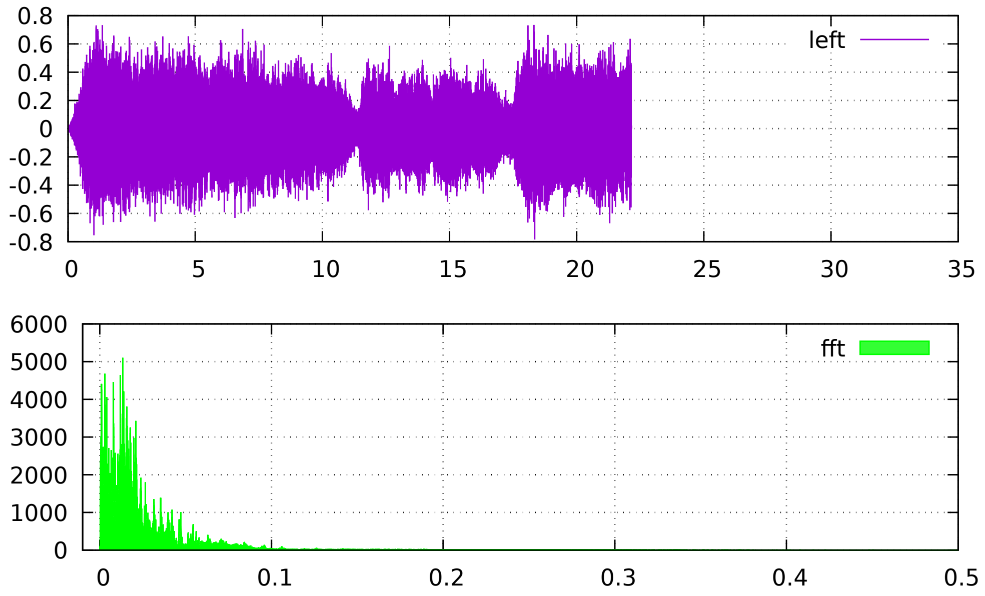

Figure 4.7: Plot generated by the following: run the commands at the shell to generate the data, then plot the data with with “`gnuplot gloria-excelsis-deo.gp`”

commands to generate data

```
wget -q --continue https://upload.wikimedia.org/wikipedia/en/e/ea/Angels_We_Have_Heard_on_High%2C_sung_by_the_Mormon_Tabe
ffmpeg -n -i gloria.ogg gloria.aiff
sox -q gloria.aiff -t dat gloria.dat
./simple-fft.py gloria.dat 1 gloria-fft.dat
```

gnuplot instructions

```
set multi layout 2, 1
set grid
set style fill solid 0.8
plot 'gloria.dat' using 1:2 with lines t 'left'
plot [-0.01:] 'gloria-fft.dat' using 1:3 with boxes \
     lt rgb "green" t 'fft'
```

4.2.6 Create your own audio clip and analyze it

Record some sound and analyze it.

The program `sox`, which we used to do audio format conversion, comes with two other programs `rec` (to record from your computer's microphone) and `play` to play those files back.

To try it out run the following:

```
$ rec myvoice.dat
## speak for a second or two into the microphone, then hit control-C
$ play myvoice.dat
$ ./simple-fft.py myvoice.dat 1 myvoice-fft.dat
$ gnuplot
gnuplot> set multi layout 2,1
gnuplot> plot 'myvoice.dat' using 1:2 with lines
gnuplot> plot 'myvoice-fft.dat' using 1:3 with boxes
```

Now try doing this again for different things you can record with your microphone. If you have a tuning fork, tap it and then rest it on a guitar's soundboard and record that, see if you get something similar to what we saw in Section 4.2.1. If you have a musical instrument, try recording an A note or an F note and compare them to what we saw in Section 4.2.3.

4.3 Picking out frequencies from a toy signal

In Section 1.4 we generated a simple sin wave. The fourier analysis of that should offer a clear single peak, as we can see with

```
$ ./make-simple-wave.py > simplewave.dat
$ ./simple-fft.py simplewave.dat 1 simplewave-fft.dat
$ gnuplot
gnuplot> set multi layout 2,1
gnuplot> plot 'simplewave.dat' using 1:2 with lines
gnuplot> plot 'simplewave-fft.dat' using 1:3 with boxes
```

Now let us generate a slightly more complex signal, one that will add two sin waves together. We will do it to mimic the temperature over a couple of years, like what we retrieved in Chapter 2. Start by making a copy of the program `generate-sin-cleaner.py` to `generate-two-sin.py`, and then we will try adding two different things to it: *(a)* a higher-frequency sin wave, analogous to the daily fluctuations in temperature on top of the yearly fluctuations, and *(b)* a certain amount of white noise.

The Fourier transform of these two different additions should be instructive. Both will *look the same:* a lot of spikes on top of the annual period. They will look different when you zoom in (as in Figure 2.2), but the Fourier transform will also allow us to pick out a strong difference.

Modify `generate-two-sin.py` to look like this:

```
#! /usr/bin/env python3

"""Generate samples of a two sin() waves and saves them to a file.
The file has two columns of data separated by white space. To run the
program and plot its output try:

$ ./generate-two-sin.py two_sin_output.dat

$ gnuplot

gnuplot> plot 'sin_output.dat' using 1:2 with lines
"""

import math

years = 2.5
minutes_per_day = 24*60
minutes_per_year = 365.25 * minutes_per_day
n_minutes = int(years * minutes_per_year)
```

```
annual_avg_temp = 21.0 ## degC
annual_temp_range = 40
daily_temp_range = 10

## zero point is
#for minutes in range(0, n_minutes, 5): ## 5-minute intervals
for minutes in range(0, n_minutes, 45): ## 45-minute intervals
    annual_term = math.sin(2*math.pi * minutes/minutes_per_year)
    daily_term = math.sin(2*math.pi * minutes/minutes_per_day)
    temp = (annual_avg_temp + annual_term*annual_temp_range/2
            + daily_term*daily_temp_range/2)
    print('%g   %g    %g    %g'
          % (float(minutes)/minutes_per_day, annual_term, daily_term, temp))
```

Listing 4.3: Generate two sin waves: `generate-two-sin.py`

The results are shown in Figure 4.8: we see two peaks, one for the "annual" cycle, the other for the higher frequency "daily" cycle. There are many points, so the peaks are not that thick, but they are tall and well-defined.

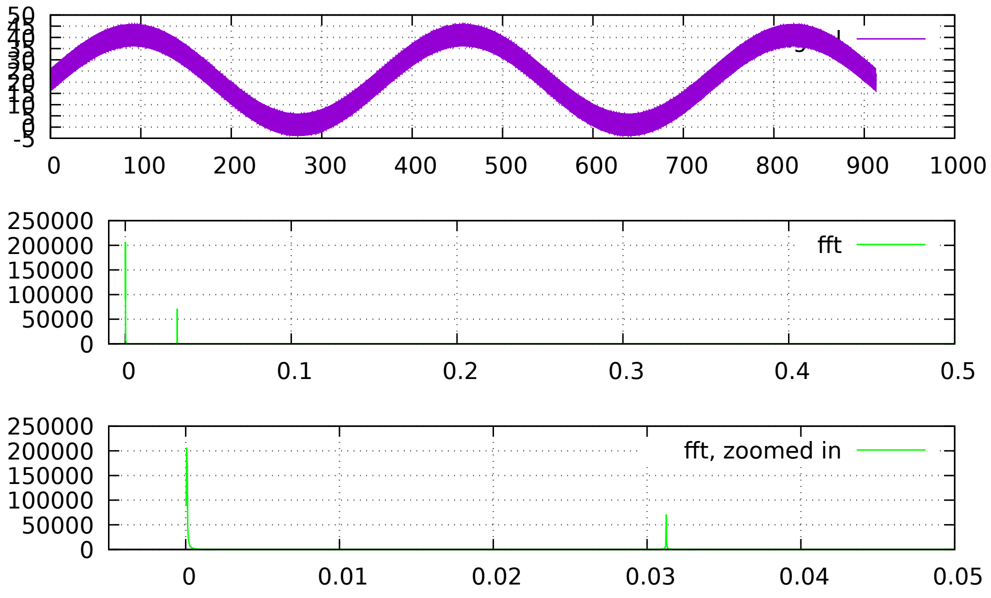

Figure 4.8: Plot generated by the following: run the commands at the shell to generate the data, then plot the data with with "`gnuplot two-sin-fft.gp`"

commands to generate data

```
./generate-two-sin.py > two_sin_output.dat
./simple-fft.py two_sin_output.dat 3 two_sin_fft.dat
```

gnuplot instructions

```
set multi layout 3, 1
set grid
set style fill solid
plot 'two_sin_output.dat' using 1:4 with lines t 'signal'
plot [-0.01:] 'two_sin_fft.dat' using 1:3 \
     with lines lt rgb "green" t 'fft'
plot [-0.005:0.05] 'two_sin_fft.dat' using 1:3 \
     with lines lt rgb "green" t 'fft, zoomed in'
```

Then we do the same thing with a *noisy* sin wave. The program looks like this:

```python
#! /usr/bin/env python3

import math
import random

years = 2.5
minutes_per_day = 24*60
minutes_per_year = 365.25 * minutes_per_day
n_minutes = int(years * minutes_per_year)
annual_avg_temp = 21.0 ## degC
annual_temp_range = 40
daily_temp_range = 10

## zero point is
#for minutes in range(0, n_minutes, 5): ## 5-minute intervals
for minutes in range(0, n_minutes, 45): ## 5-minute intervals
    annual_term = math.sin(2*math.pi * minutes/minutes_per_year)
    noisy_term = -0.5 + random.uniform(-1.0, 1.0)
    temp = (annual_avg_temp + annual_term*annual_temp_range/2
            + noisy_term*daily_temp_range/2)
    print('%g   %g   %g   %g'
          % (float(minutes)/minutes_per_day, annual_term, noisy_term, temp))
```

Listing 4.4: Generate a noisy sin: `make-noisy-wave.py`

The noisy sin wave and its Fourier transform are shown in Figure 4.9, and you can see that although the signal is hard to tell apart from the double sin wave, the spectrum is clear: there is only one significant peak (from the "yearly" sin wave), and then there is a scattering of noisy bits in the spectrum, which we expect when the signal is noisy instead of being a pure sin wave.

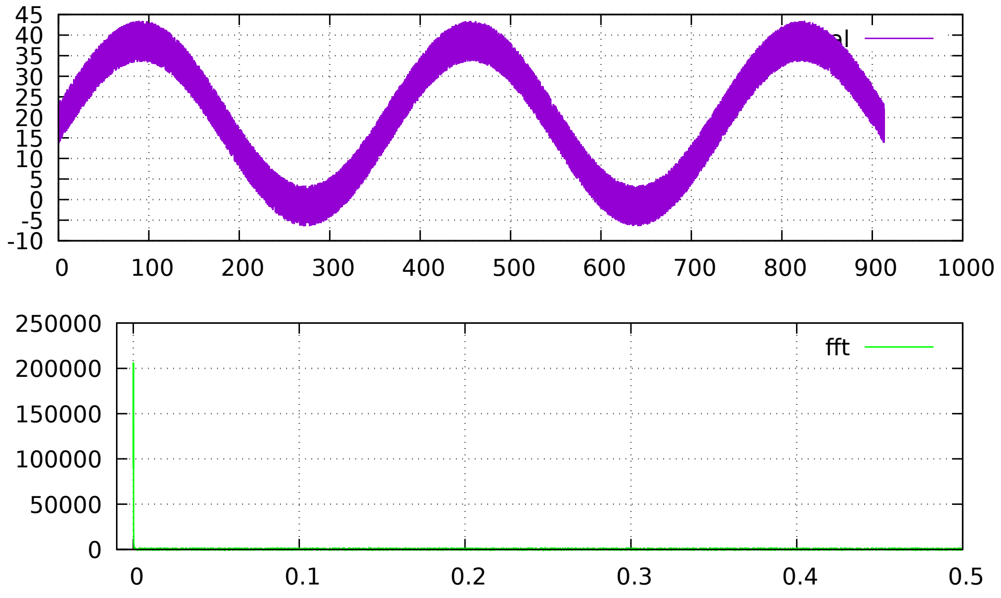

Figure 4.9: Plot generated by the following: run the commands at the shell to generate the data, then plot the data with with "`gnuplot noisy-sin-fft.gp`"

commands to generate data

```
./make-noisy-wave.py > noisy_sin_output.dat
./simple-fft.py noisy_sin_output.dat 3 noisy_sin_fft.dat
```

gnuplot instructions

```
set multi layout 2, 1
set grid
set style fill solid
plot 'noisy_sin_output.dat' using 1:4 with lines t 'signal'
plot [-0.01:] 'noisy_sin_fft.dat' using 1:3 \
     with lines lt rgb "green" t 'fft'
```

4.4 EXTRA: Fourier analysis of the temperature data

Our final step is to look at the temperature data we downloaded in Chapter 2. Unlike Section 4.3, this example will use *real* data from the weather station near Las Cruces.

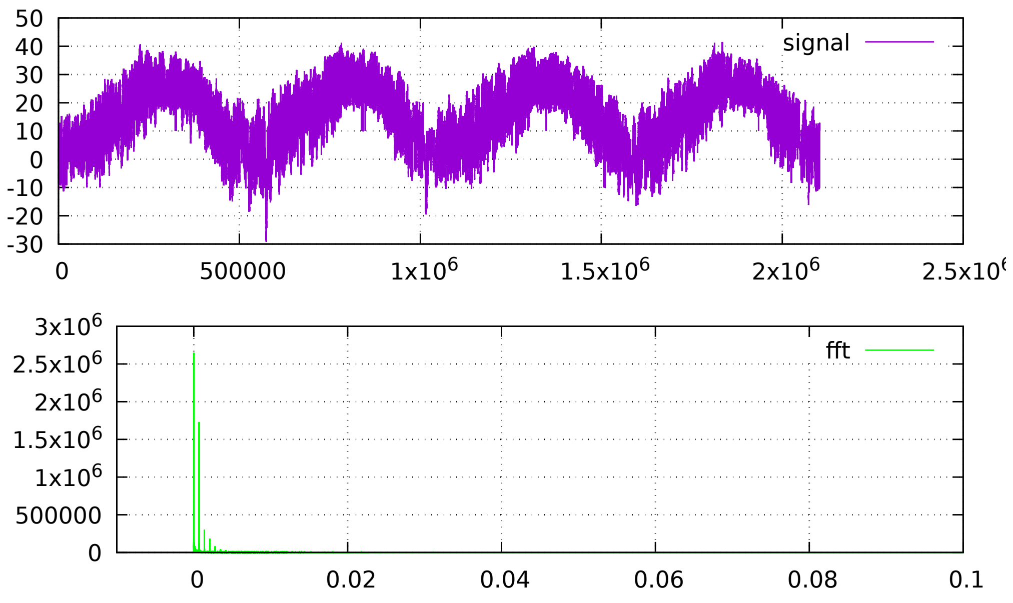

Figure 4.10: Plot generated by the following: run the commands at the shell to generate the data, then plot the data with with "`gnuplot temperature-fft.gp`"

commands to generate data

```
./retrieve-temperature-first.py
./temperature-fft.py Las_Cruces
```

gnuplot instructions

```
set multi layout 2, 1
set grid
set style fill solid
plot 'temperatures_Las_Cruces.dat' using 1:2 with lines t 'signal'
plot [-0.01:] 'temperatures_Las_Cruces-fft.dat' using 1:3 \
     with lines lt rgb "green" t 'fft'
```

The Fourier transform of the temperature is shown in Figure 4.10. We see something analogous to what we saw with the toy model which simulated temperature by adding two sin waves.

Appendix A

GNU Free Documentation License

Preamble

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "**Document**", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "**you**". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "**Modified Version**" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "**Secondary Section**" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "**Invariant Sections**" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "**Cover Texts**" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "**Transparent**" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "**Opaque**".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated

K. For any section Entitled “Acknowledgements” or “Dedications”, Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled “Endorsements”. Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled “Endorsements” or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled “Endorsements”, provided it contains nothing but endorsements of your Modified Version by various parties–for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled “History” in the various original documents, forming one section Entitled “History”; likewise combine any sections Entitled “Acknowledgements”, and any sections Entitled “Dedications”. You must delete all sections Entitled “Endorsements”.

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an “aggregate” if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled “Acknowledgements”, “Dedications”, or “History”, the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

Bibliography

Backus, J. *Musical Note to Frequency Conversion Chart.* 1977. URL: `%5Curl%7Bhttp://www.audiology.org/sites/default/files/ChasinConversionChart.pdf%7D` (visited on 05/06/2016).

Galassi, Mark. *Hacking Camp Teacher's Manual.* 2015.

Pinker, Steven. *The better angels of our nature: Why violence has declined.* Penguin, 2011.

Reinhart, Carmen M and Kenneth S Rogoff. "Growth in a time of debt (digest summary)". In: *American Economic Review* 100.2 (2010), pp. 573–578.