Iain Gray

Snake Charming—The Musical Python

EXTRAS ONLINE

Springer

Snake Charming—The Musical Python

Iain Gray

Snake Charming—The Musical Python

Springer

Iain Gray
AFIMA—Associate Fellow of the Institute
of Mathematics and its Applications
Southend-on-Sea
UK

Additional material to this book can be downloaded from http://extras.springer.com.

ISBN 978-3-319-60659-0 ISBN 978-3-319-60660-6 (eBook)
DOI 10.1007/978-3-319-60660-6

Library of Congress Control Number: 2017947760

Printed on acid-free paper

This Springer imprint is published by Springer Nature
The registered company is Springer International Publishing AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

To Jim Martin, a fellow traveller on the road to Parnassus, this book is dedicated in friendship

Preface

Intended Audience

This book is an innovative introduction to Python and its audio-visual capabilities for beginning programmers, a resource for expert programmers and of interest to anyone involved in music. It is structured around four extensible, audio-visual projects on music and sound. The beginner will appreciate the 'need to know' basis of the presentation of Python for each project. Expert programmers will be able to go straight to the project code, run it and then extend it as they see fit. Musically interested readers will enjoy the historical and theoretical material at the beginning of each project, and it may even tempt them to try some coding—it is not too difficult! The projects are all self-contained but can be extended to incorporate aspects of the others. Above all, the book is suited for self-study, which should be playful (pun intended)!

Prerequisites

A minimal understanding of your computer's operating system is assumed. In particular, you should be able to

1. find, create and delete files or folders within the filing system
2. open, close, save and print files
3. run program files by opening or 'double clicking' them
4. that's all!

Typography

Normal text will appear in a Times Roman font, whereas Python code will appear in a monospaced `courier` font. The first use of a new program construct will be highlighted as `construct`. Control and modifier keys will appear between angle brackets and in a bold typeface as in **<return>** or **<enter>**.

A Note on the Code

The code largely complies with functional and procedural paradigms, and a reference to object-oriented Python is contained in Appendix. In the **New Language Features** sections, it is intentionally repetitive and uses many print statements, rather than breakpoints, to explain its operation. The overall driver has been simplicity of exposition. After all as Albert Einstein said 'Everything should be made as simple as possible, but not simpler'.

Structure of Book

In order to achieve expertise in the audio-visual and animation capabilities of Python, four major projects in music and sound will be developed. The book is divided into six parts

1. Part I—Installation, shell, editor, Python syntax and package management
2. Part II—Sound visualisation
3. Part III—Sound creation
4. Part IV—Harmonic visualisation
5. Part V—Composition
6. Part VI—Future development

Part II–Part V comprise the projects and are each split into background and coding chapters.

Chapter 14 contains suggestions and guidelines for extending the projects. Finally, all the internet links and bibliography are in Appendix.

This book and its projects loosely parallel the historical development of music; from Rhythm II through Melody III to Harmony IV, and finally on to Composition V.

Edinburgh, UK
2017

Iain Gray

Acknowledgements

A book like this does not spring fully armed like Athene from the thigh of Zeus, but relies on the active involvement of several groups of people

1. my friends and colleagues in Radar Systems Design for discussions over several decades.
2. my parents for constant support and encouragement.
3. my editors at Springer, Beverley and Nancy, for their courtesy and patience.

Contents

Part I Snake in the Grass—Python and Its Environment

1 Installing Python 3

2 The Python Shell—IDLE 7
- 2.1 Basic Python Syntax 8
  - 2.1.1 Comments 8
  - 2.1.2 Indentation and Block Structure 8
  - 2.1.3 Input and Output 8
  - 2.1.4 Declaration of Simple Types and Type Casting 9
  - 2.1.5 Arithmetic Operators and Precedence 9
  - 2.1.6 Conditional Expressions, Relational and Logical Operators 9
  - 2.1.7 Conditional Statements 10
  - 2.1.8 Looping Statements 10
- 2.2 Entering Python Code 11
  - 2.2.1 The Python Interpreter 11
  - 2.2.2 The Python Editor 12

3 Package Management 15
- 3.1 Anaconda 15
  - 3.1.1 Installing Anaconda 16
  - 3.1.2 Using Anaconda 17
- 3.2 Alternatives 18

4 Audacity® 19
- 4.1 Installing 19
- 4.2 Using 19

Part II Banging the Drum—Visualising Sound

5 Mark Kac (1914 to 1984) 23
5.1 Hearing the Shape of a Drum 24
5.2 Riding the Waves—Bessel Functions 24
5.3 Vibrating Plates—Chladni and Germain 25
5.4 Drumhead Modes 26

6 Project Code 29
6.1 New Language Features 29
6.1.1 Project Header 29
6.1.2 Plotting Bessel Functions 30
6.1.3 Graphing in 3D 31
6.1.4 Animation of Square Wave from Summing Sinusoids 32
6.1.5 Animating in 3D, for Wave Propagation Along an Axis 33
6.2 The Code 34
6.2.1 Project Header 34

Part III Heat in the Desert—Sculpting Sound

7 Joseph Fourier (1768 to 1830) 37
7.1 The Army of Egypt 37
7.2 Feeling the Heat—Fourier Transforms 38
7.3 Chasing Rainbows—Frequency Spectra 38

8 Bob Moog (1934 to 2005) 39
8.1 Analogue Additive Synthesis 40
8.2 Analogue Subtractive Synthesis 40
8.2.1 Oscillators 41
8.2.2 Filters 45
8.2.3 Amplifier 46
8.2.4 Envelope Generation 46
8.2.5 Modulation 47

9 Project Code 49
9.1 New Language Features 49
9.1.1 Using Tkinter 49
9.1.2 Project Header 50
9.1.3 Sound Storage 50
9.1.4 Harmonic Analysis 51
9.1.5 Oscillators and Mixer 51
9.1.6 Low Pass Filtering 55
9.1.7 Implement Butterworth Low Pass Filter 57
9.1.8 Amplitude Envelope Generation 58

9.1.9 Low Frequency Oscillator and Modulation 60
9.1.10 Analysis Displays 61
9.1.11 Graphical User Interface 63
9.1.12 GUI Support for Mixer Button 64
9.2 The Code 65
9.2.1 Project Header. 65
9.2.2 Oscillator and Mixer 65
9.2.3 Filter 66
9.2.4 Amplitude Envelope Generator 67
9.2.5 Modulator 67
9.2.6 Amplifier. 68
9.2.7 Displays and Output 68
9.2.8 User Interface 69

Part IV The Harmonograph—Victorian Pendulum Toy

10 Hugh Blackburn (1823 to 1909) 73
10.1 Motion of a Damped Pendulum 74
10.2 Blackburn's Double Pendulum 75
10.3 Harmonic Ratios—The Lateral Harmonograph 76
10.4 Parallels—Bowditch and Lissajous 76
10.5 Of Gears and Motors—The Pintograph 78

11 Project Code 79
11.1 New Language Features 79
11.1.1 Lissajous' Figures 79
11.1.2 Damped Orthogonal Pendulums 81
11.1.3 Harmonic Ratios as Fractions 82
11.1.4 User Interface 83
11.1.5 Project Header. 83
11.2 The Code 83
11.2.1 Project Header. 83
11.2.2 Orthogonal Polynomials 84
11.2.3 User Interface 85

Part V Counterpoint à la Mode—Composing Music

12 Johann Joseph Fux (1660 to 1741). 89
12.1 Gradus Ad Parnassum—Counterpoint. 90
12.1.1 Melody—Direct, Contrary and Oblique Motion 90
12.1.2 Harmony—Consonance and Dissonance 90
12.1.3 Species Counterpoint. 91
12.1.4 Modal Music. 91
12.2 Strict Rules Allow Freedom of Composition 92

13 Project Code . . . 93
13.1 The Colours of Noise . . . 93
13.2 New Language Features . . . 95
13.2.1 Project Header . . . 96
13.2.2 Sound Storage . . . 96
13.2.3 MIDI—Musical Instrument Digital Interface . . . 96
13.2.4 Composing the Cantus Firmus with Pink Noise Generation . . . 97
13.2.5 Sound Generation . . . 98
13.2.6 Sound Output of Cantus Firmus . . . 102
13.2.7 The Rules in Python . . . 103
13.2.8 Output of Whole, Half and Quarter Notes for Counterpoint . . . 104
13.2.9 User Interface and Sound . . . 104
13.3 The Code . . . 105
13.3.1 Project Header . . . 105
13.3.2 Generating the Cantus Firmus with Pink Noise . . . 106
13.3.3 Cantus Firmus Dynamics . . . 106
13.3.4 Rules of Counterpoint . . . 107
13.3.5 Converting MIDI to Frequency . . . 108
13.3.6 Handling Whole, Half and Quarter Notes . . . 108
13.3.7 Stereo Output . . . 109
13.3.8 User Interface . . . 109

Part VI On Safari

14 Where Next? . . . 113
14.1 Generic Header . . . 113
14.2 Part II—Visualising Sound . . . 114
14.3 Part III—Creating Sound . . . 114
14.4 Part IV—Visualising Harmony . . . 115
14.5 Part V—Composition . . . 116

Curriculum Vitae . . . 117

Appendix: Internet Links . . . 119

Part I
Snake in the Grass—Python and Its Environment

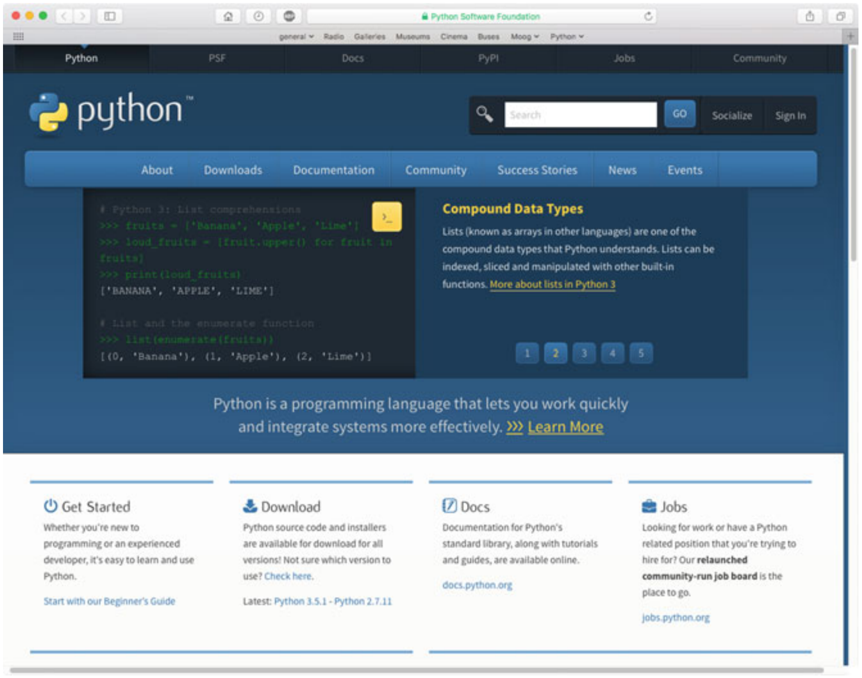
Chapter 1
Installing Python

“Latet anguis in herba” Virgil, Eclogues III.

I. Gray, *Snake Charming—The Musical Python*,
DOI 10.1007/978-3-319-60660-6_1

Python is a simple and elegant language that is easy to learn and install on any platform. The language is cross-platform and truly multi-paradigm embodying functional, imperative and object-oriented features. A simple development environment IDLE is also installed with Python allowing interactive programming and syntax aware code editing. Sophisticated package managers allow easy access to pre-made scientific and graphics libraries amongst many others.

On visiting the Python homepage https://www.python.org you will see.


Python Software Foundation
general Radio Galleries Museums Cinema Buses Moog Python
Python PSF Docs PyPI Jobs Community
python™
Search GO Socialize Sign In
About Downloads Documentation Community Success Stories News Events
1 2 3 4 5

```
# Python 3: List comprehensions
>>> fruits = ['Banana', 'Apple', 'Lime']
>>> loud_fruits = [fruit.upper() for fruit in
fruits]
>>> print(loud_fruits)
['BANANA', 'APPLE', 'LIME']

# List and the enumerate function
>>> list(enumerate(fruits))
[(0, 'Banana'), (1, 'Apple'), (2, 'Lime')]
```

Compound Data Types

Lists (known as arrays in other languages) are one of the compound data types that Python understands. Lists can be indexed, sliced and manipulated with other built-in functions. More about lists in Python 3

Python is a programming language that lets you work quickly and integrate systems more effectively. >>> Learn More

Get Started

Whether you're new to programming or an experienced developer, it's easy to learn and use Python.

Start with our Beginner's Guide

Download

Python source code and installers are available for download for all versions! Not sure which version to use? Check here.

Latest: Python 3.5.1 - Python 2.7.11

Docs

Documentation for Python's standard library, along with tutorials and guides, are available online.

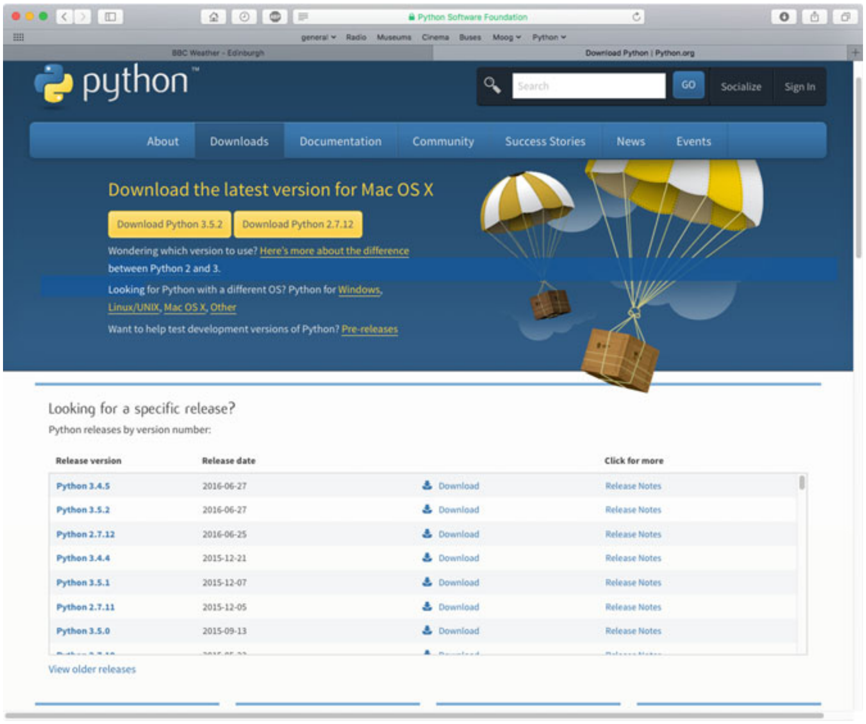
docs.python.org

Jobs

Looking for work or have a Python related position that you're trying to hire for? Our **relaunched community-run job board** is the place to go.

jobs.python.org

Clicking on the ‘Downloads’ tab followed by ‘All releases’ in the left hand sidebar you will then see



Download the latest version for Mac OS X

Wondering which version to use? Here's more about the difference between Python 2 and 3.

Looking for Python with a different OS? Python for Windows, Linux/UNIX, Mac OS X, Other

Want to help test development versions of Python? Pre-releases

Looking for a specific release?

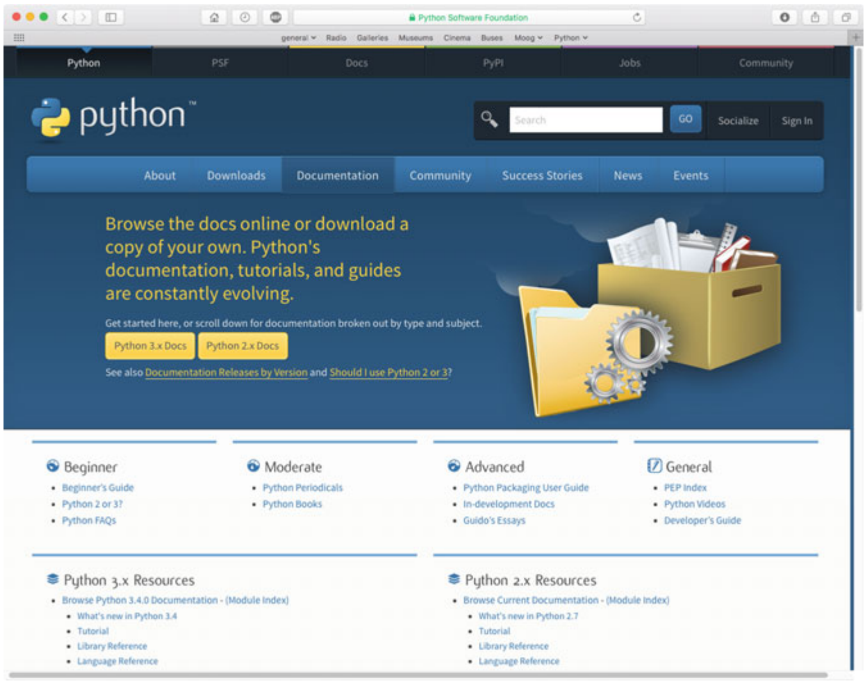
Python releases by version number:

Release version	Release date		Click for more
Python 3.4.5	2016-06-27	Download	Release Notes
Python 3.5.2	2016-06-27	Download	Release Notes
Python 2.7.12	2016-06-25	Download	Release Notes
Python 3.4.4	2015-12-21	Download	Release Notes
Python 3.5.1	2015-12-07	Download	Release Notes
Python 2.7.11	2015-12-05	Download	Release Notes
Python 3.5.0	2015-09-13	Download	Release Notes

View older releases

so click on ‘Download Python 3.5.2’ to have a graphical installer loaded on your machine. Open this installer by double clicking on it and that is all—light blue touch paper and retire! The homepage identifies your target architecture and will install the relevant version of Python, and as the language is machine independent Python source code will run on any supported architecture. Note that Python 3.5.2 was the latest version at time of press but the installation procedure above will always obtain the latest version.

Finally clicking on the ‘Documentation’ tab followed by ‘Docs’ in the left hand sidebar will bring you to the documents page


Python Software Foundation
general Radio Galleries Museums Cinema Buses Moog Python
Python PSF Docs PyPI Jobs Community
python™
Search
GO
Socialize
Sign In
About Downloads Documentation Community Success Stories News Events
Python 3.x Docs
Python 2.x Docs

Browse the docs online or download a copy of your own. Python's documentation, tutorials, and guides are constantly evolving.

Get started here, or scroll down for documentation broken out by type and subject.

See also Documentation Releases by Version and Should I use Python 2 or 3?

Beginner
- Beginner's Guide
- Python 2 or 3?
- Python FAQs

Moderate
- Python Periodicals
- Python Books

Advanced
- Python Packaging User Guide
- In-development Docs
- Guido's Essays

General
- PEP Index
- Python Videos
- Developer's Guide

Python 3.x Resources
- Browse Python 3.4.0 Documentation - (Module Index)
  - What's new in Python 3.4
  - Tutorial
  - Library Reference
  - Language Reference

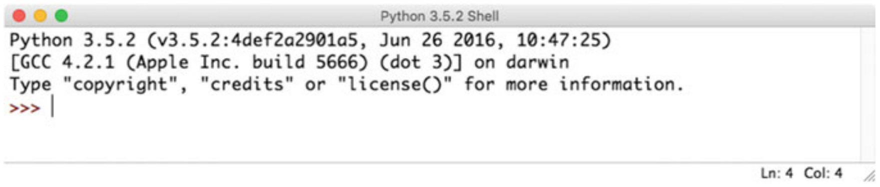
Python 2.x Resources
- Browse Current Documentation - (Module Index)
  - What's new in Python 2.7
  - Tutorial
  - Library Reference
  - Language Reference

from where you can access tutorials and the main language references online.

Chapter 2
The Python Shell—IDLE

IDLE, the Integrated Development and Learning Environment, is a shell that comes bundled with the Python distribution on download.

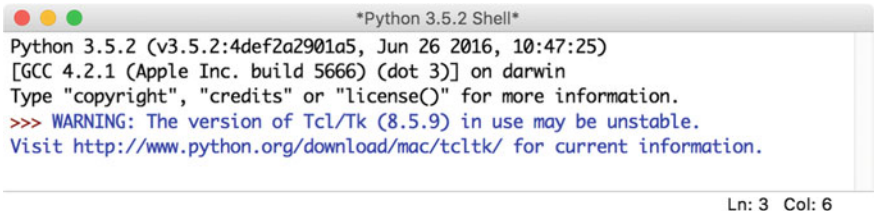
Locate the Python 3.5 folder, open it, find IDLE and open it seeing a large empty window headed by



```
Python 3.5.2 (v3.5.2:4def2a2901a5, Jun 26 2016, 10:47:25)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "copyright", "credits" or "license()" for more information.
>>> |
```

the final >>> prompt is where you can enter code interactively for the Python interpreter.

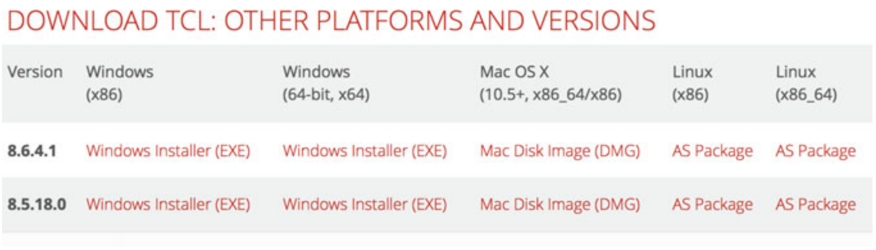
If on any platform you see the following warning message



```
Python 3.5.2 (v3.5.2:4def2a2901a5, Jun 26 2016, 10:47:25)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "copyright", "credits" or "license()" for more information.
>>> WARNING: The version of Tcl/Tk (8.5.9) in use may be unstable.
Visit http://www.python.org/download/mac/tcltk/ for current information.
```

it is safe to ignore this and type <**return**> or <**enter**> carrying on with the rest of the chapter. You will only be using 'Tcl/Tk' within the 'spyder' environment of Chap. 3. To clear this warning you have to install the latest Tcl/Tk 8.5 release which was 8.5.18.0 at the time of writing. Visit the ActiveState Tcl/Tk download page at http://www.activestate.com/activetcl/downloads and scroll down to

I. Gray, *Snake Charming—The Musical Python*,
DOI 10.1007/978-3-319-60660-6_2



DOWNLOAD TCL: OTHER PLATFORMS AND VERSIONS

Version	Windows (x86)	Windows (64-bit, x64)	Mac OS X (10.5+, x86_64/x86)	Linux (x86)	Linux (x86_64)
8.6.4.1	Windows Installer (EXE)	Windows Installer (EXE)	Mac Disk Image (DMG)	AS Package	AS Package
8.5.18.0	Windows Installer (EXE)	Windows Installer (EXE)	Mac Disk Image (DMG)	AS Package	AS Package

from where you can download the latest 8.5 version to install for your platform.

2.1 Basic Python Syntax

Python itself has an ‘official’ tutorial accessible as described in Chap. 1 or as in Chap. 3. The aim here is slightly less lofty and is intended as a ‘get you started’ subset of Python. More advanced concepts such as functions or datatypes such as lists will be introduced in the projects as and when required.

2.1.1 Comments

Comments to be ignored by the interpreter are introduced by the # (hash) character (<**option**> or <**alt**> + 3 on many keyboards). Multi-line comments can be delimited (before and after) by ”’ (three single quotes).

2.1.2 Indentation and Block Structure

Python is block structured but uses code indentation, rather begin...end, to delimit related blocks of code.

2.1.3 Input and Output

The most basic functions are input() and print().

2.1.4 Declaration of Simple Types and Type Casting

Only the numeric type of int and float will be used here: an int holds an integer value and has no decimal point: a float hold a floating point value and has a decimal point. The two types have different internal representations. Before being used an alphanumeric variable must be declared to be of a particular type by

1. assigning it to a constant as in a = 0.0
2. making multiple assignments as in a, b, c = 0.0, 1, 2
3. assigning it to the value of an expression as in a = b/c, where b and c have previously been declared.

Variables of one type may be cast as the other by using the int() and float() functions.

2.1.5 Arithmetic Operators and Precedence

The basic arithmetic operators are $+$, $-$, $*$, / supplemented by ** for exponentiation, // for floor (integer) division and % for remainder. Parsing of arithmetic expressions is carried out left to right obeying the following order of precedence

1. (highest) ** and unary minus but see note below
2. $*, /, //, \%$
3. (lowest) $+, -$

Note that ** is higher than left unary minus, ** is lower than a right unary minus in terms of precedence.

This order of precedence may be changed by using parentheses ().

2.1.6 Conditional Expressions, Relational and Logical Operators

Python has a cast of ‘all the usual suspects’ for its relational operators, which are

1. $==$ equals. Note that a single $=$ is only used for declaration or assignment
2. $! =$ not equals
3. $>$ greater than
4. $>=$ greater than or equals
5. $<$ less than
6. $<=$ less than or equals.

These all return Boolean values of True or False and are ranked below ‘$+, -$’ in order of precedence. They can be chained together to form multiple relational expressions such as $x < y < z$.

Logical operators can be used to combine such Boolean values into composite conditional expressions with a value of True or False. They rank lower than the relational operator and have the following order of precedence

1. **not** expr—True if expr is False, False otherwise
2. expr1 **and** expr2—True if both expr1 and expr2 are True, False otherwise
3. expr1 **or** expr1—True if either expr1 or expr2 is True, False otherwise.

Conditional expressions can thus be made of one, or more, relational expressions tied together with logical operators.

2.1.7 Conditional Statements

Multiple line statements are much more easily entered within an Editor shell (see later section).

Conditional statements are those which allow the selection of a consequence or alternative statement depend on the evaluation of a conditional expression. They come in three flavours which are best illustrated by examples.

1. conditional statements have a simple structure and are most often used for boolean assignments, thus `out=x and not y or not x and y`
2. **inline if** is used for conditional assignment and has a simple structure, thus `abs=-x if x<0 else x`
3. **if, elif, else** conditional statements will perform blocks of statements, after each**:**, dependent on the result of the conditional expression. **elif** short for **else if** allows for nested conditional statements. As described earlier indentation is used to distinguish blocks. For example

```
if number<0:
    print('negative')
elif number>0:
    print ('positive')
else:
    print('zero')
```

2.1.8 Looping Statements

Multiple line statements are much more easily entered within an Editor shell (see later section).

Looping statements are those which allow the controlled repetition of a statement or group of statements. They again come in two flavours which are best illustrated by examples. The examples are in the Python editor section and cover both simple versions of **for** loops and **while** loops. No attempt has been made to explain list types, or **in** (membership) operators as such details are deferred to later Project Code chapters.

2.2 Entering Python Code

IDLE supports two types of shell

1. the Python shell which support interactive development via the Python interpreter
2. one or more Editor shells which allow you to edit and save Python code.

2.2.1 The Python Interpreter

Type the following into the IDLE Python interpreter shell, (comments are optional), followed by <**return**> or <**enter**> for each line and observe the output.

```
#multiple declarations
a,b=355,113
#multiple assignments
p,q,r=a/b,a//b,a%b
#multiple prints
print(p,q,r)
s=float(a)+r/b
print(s)
p==s
#generally unsafe to use == (equality) on floats
#Boolean declarations
x,y,out=False,True,False
#'exclusive or' as a conditional expression
out=x and not y or not x and y
#'abs' as an inline if statement
abs=-x if x<0 else x
```

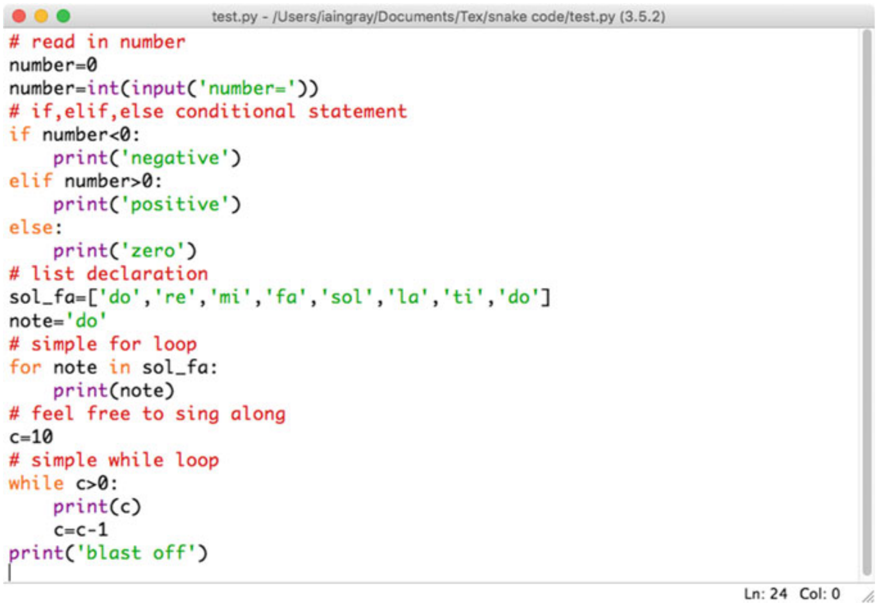
2.2.2 The Python Editor

Unfortunately the Python interpreter shell vanishes on quitting IDLE. to recall your work you must create a new editor shell, type in your program and save it with a .py extension. Then on restarting IDLE open your file to run it, or ‘run module’ in the Python shell via the Run menu in the Editor shell. The editor offers many features like automatic syntax highlighting, auto-completion of language words, auto-indentation of blocks and highlighting parenthesised expressions.

Open a new Editor shell, and type the following into the IDLE editor shell, (comments are optional). Save your work in ‘test.py’, reopen as described above and observe the output. Note that although strings are used this is not a ‘hello world’ program

```
# read in number
number=0
number=int(input('number='))
# if, elif, else conditional statement
if number<0:
    print('negative')
elif number>0:
    print ('positive')
else:
    print('zero')
# list declaration
sol_fa=['do','re','mi','fa','sol','la','ti','do']
note='do'
# simple for loop
for note in sol_fa:
    print(note)
# feel free to sing along
c=10
# simple while loop
while c>o:
    print(c)
    c=c−1
print('blast off')
```

Finally the syntax highlighted IDLE editor window for ‘test.py’ looks like


test.py - /Users/iaingray/Documents/Tex/snake code/test.py (3.5.2)
Ln: 24 Col: 0

```
# read in number
number=0
number=int(input('number='))
# if,elif,else conditional statement
if number<0:
    print('negative')
elif number>0:
    print('positive')
else:
    print('zero')
# list declaration
sol_fa=['do','re','mi','fa','sol','la','ti','do']
note='do'
# simple for loop
for note in sol_fa:
    print(note)
# feel free to sing along
c=10
# simple while loop
while c>0:
    print(c)
    c=c-1
print('blast off')
```

Chapter 3
Package Management

Over the years the Python community has developed many library packages to expand and enhance the basic Python implementation. Of most relevance for the projects in this book are the following three packages

1. NumPy—a specialised numerical package adding random numbers, Fourier transforms and linear algebra over multi-dimensional matrices
2. SciPy—a stack of packages supporting scientific applications
3. Matplotlib—support for two and three dimensional graph plotting and visualisation

Note that packages are also referred to as libraries and modules elsewhere, but the generic term package will be retained throughout this book.

As well as the packages it is very useful to have a package manager to automatically install them and keep them up to date.

3.1 Anaconda

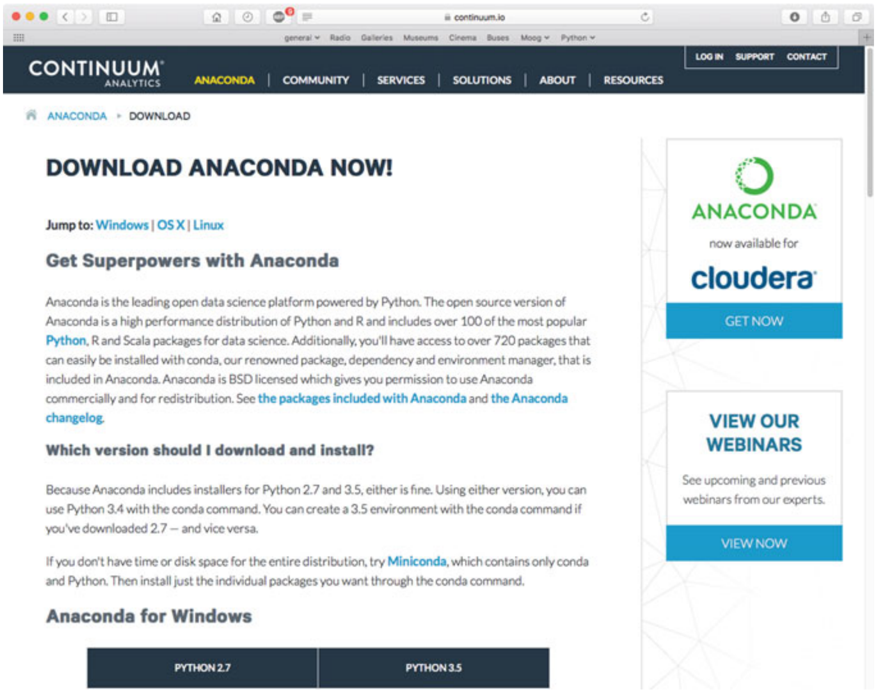
Anaconda provides the most comprehensive package manager currently available for Python as well as giving direct access to supporting applications and documentation. All the packages are automatically installed and updated by Anaconda.

During the production of this book Python was upgraded from version 3.5.1 to version 3.5.2, Anaconda automatically upgraded on being started up.

I. Gray, *Snake Charming—The Musical Python*,
DOI 10.1007/978-3-319-60660-6_3

3.1.1 Installing Anaconda

Installing Anaconda and the Python Packages could not be simpler, visit https://www.continuum.io/downloads where you will see


continuum.io
general Radio Galleries Museums Cinema Buses Moog Python
CONTINUUM ANALYTICS
ANACONDA | COMMUNITY | SERVICES | SOLUTIONS | ABOUT | RESOURCES
LOG IN SUPPORT CONTACT
ANACONDA › DOWNLOAD
ANACONDA now available for cloudera
GET NOW
VIEW OUR WEBINARS
See upcoming and previous webinars from our experts.
VIEW NOW
PYTHON 2.7
PYTHON 3.5

DOWNLOAD ANACONDA NOW!

Jump to: Windows | OS X | Linux

Get Superpowers with Anaconda

Anaconda is the leading open data science platform powered by Python. The open source version of Anaconda is a high performance distribution of Python and R and includes over 100 of the most popular Python, R and Scala packages for data science. Additionally, you'll have access to over 720 packages that can easily be installed with conda, our renowned package, dependency and environment manager, that is included in Anaconda. Anaconda is BSD licensed which gives you permission to use Anaconda commercially and for redistribution. See the packages included with Anaconda and the Anaconda changelog.

Which version should I download and install?

Because Anaconda includes installers for Python 2.7 and 3.5, either is fine. Using either version, you can use Python 3.4 with the conda command. You can create a 3.5 environment with the conda command if you've downloaded 2.7 – and vice versa.

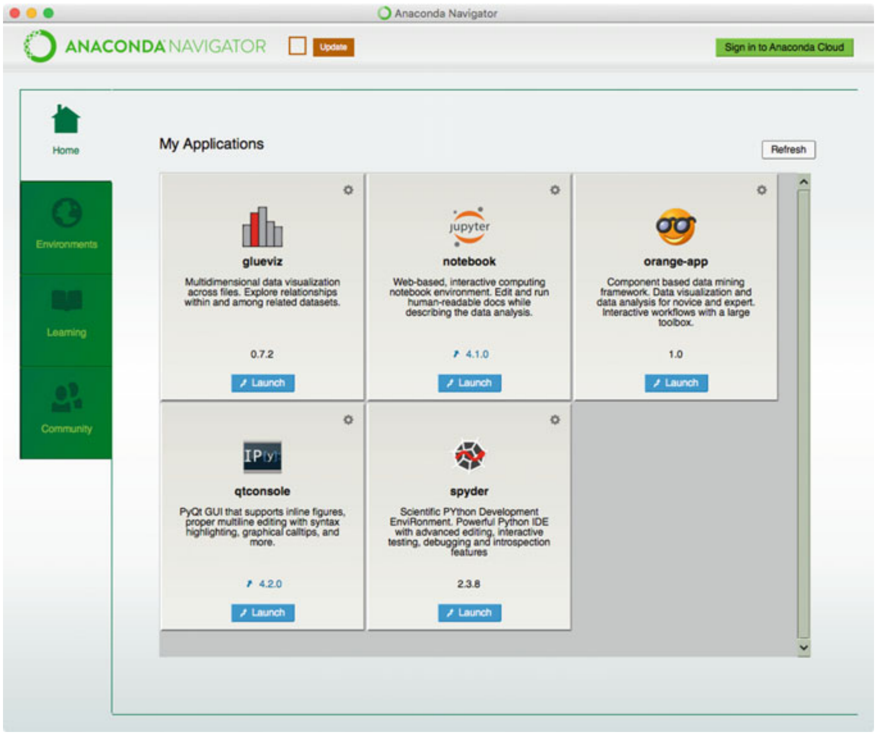
If you don't have time or disk space for the entire distribution, try Miniconda, which contains only conda and Python. Then install just the individual packages you want through the conda command.

Anaconda for Windows

Scroll down to your computer platform and choose the Python 3.5 installer, preferably if a graphical installer is available use it. Follow the on-screen instructions, light blue touch paper and retire! You will then have an Anaconda folder containing all the packages and a Navigator used to access everything in this folder.

3.1.2 Using Anaconda

Anaconda is launched by opening the Navigator to display its home screen



glueviz
Multidimensional data visualization across files. Explore relationships within and among related datasets.
0.7.2

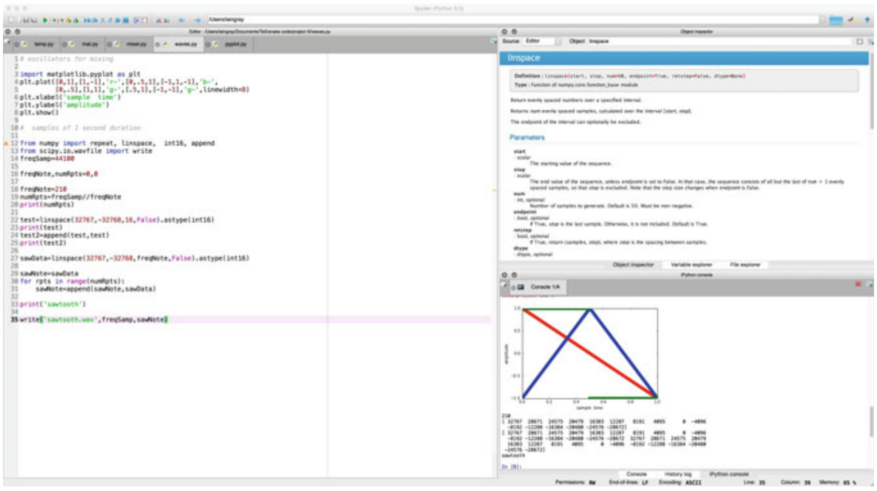
notebook
Web-based, interactive computing notebook environment. Edit and run human-readable docs while describing the data analysis.
4.1.0

orange-app
Component based data mining framework. Data visualization and data analysis for novice and expert. Interactive workflows with a large toolbox.
1.0

qtconsole
PyQt GUI that supports inline figures, proper multiline editing with syntax highlighting, graphical calltips, and more.
4.2.0

spyder
Scientific PYthon Development EnviRonment. Powerful Python IDE with advanced editing, interactive testing, debugging and introspection features
2.3.8

Note that some applications may have to be installed before being used: simply press their install buttons. The home screen will let you access documentation including tutorials via the left hand side tabs. For the projects launch Spyder and you will see



```
# oscillators for mixing

import matplotlib.pyplot as plt
plt.plot([0,1],[1,-1],'r-',[0,.5,1],[-1,1,-1],'b-',
         [0,.5],[1,1],'g-',[.5,1],[-1,-1],'g-',linewidth=8)
plt.xlabel('sample  time')
plt.ylabel('amplitude')
plt.show()

#  samples of 1 second duration

from numpy import repeat, linspace,  int16, append
from scipy.io.wavfile import write
freqSamp=44100

freqNote,numRpts=0,0

freqNote=210
numRpts=freqSamp//freqNote
print(numRpts)

test=linspace(32767,-32768,16,False).astype(int16)
print(test)
test2=append(test,test)
print(test2)

sawData=linspace(32767,-32768,freqNote,False).astype(int16)

sawNote=sawData
for rpts in range(numRpts):
    sawNote=append(sawNote,sawData)

print('sawtooth')

write('sawtooth.wav',freqSamp,sawNote)
```

```
210
[ 32767  28671  24575  20479  16383  12287   8191   4095      0  -4096
  -8192 -12288 -16384 -20480 -24576 -28672]
[ 32767  28671  24575  20479  16383  12287   8191   4095      0  -4096
  -8192 -12288 -16384 -20480 -24576 -28672  32767  28671  24575  20479
  16383  12287   8191   4095      0  -4096  -8192 -12288 -16384 -20480
 -24576 -28672]
sawtooth
```

a large syntax aware editor window on the left, an object inspector on the top right and an interactive IPython window in the bottom right. IPython is an improved IDLE with parenthesis matching, automatic completions and many other enhancements, but note that on a Mac instructions which use the <**control**> key should use the <**command**> key instead.

3.2 Alternatives

Although Anaconda provides comprehensive package management for Python your needs may be more specialised or limited and Anaconda is like 'using a sledgehammer to crack a nut'! As honest advertisers and opinion pollsters should say: "there are other products (or parties) available". These include, but are not limited to

1. PyQt—a binding of Python to the multi-platform Qt user interface
2. PySide—another binding of Python to the multi-platform Qt user interface
3. wxPython—a binding of Python to the wxWidgets C++ user interface
4. Python(x.y)—a scientific and engineering packaging of Python

The PyQt GUI already comes bundled with Anaconda. Please be aware that you may probably have have to manually upgrade any included packages within them: caveat emptor.

Chapter 4
Audacity®

Unfortunately the current 'state of the art' in Python precludes simultaneous sound output, but the use of intermediate '.wav' files facilitates analysis, recording and manipulation of these sound files. A free, open source program Audacity is very mature and stable (for all platforms) and thus is recommended for sound playback.

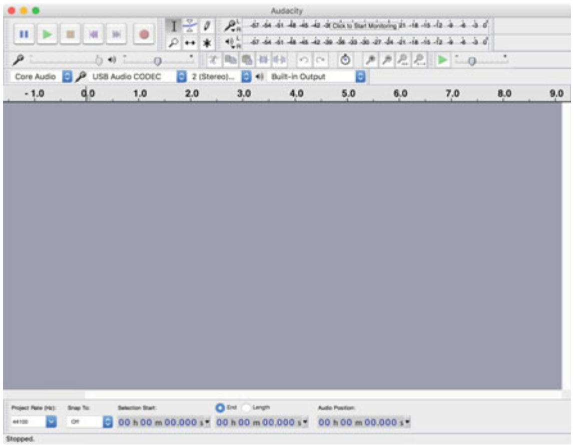
4.1 Installing

Go to http://www.audacityteam.org and download Audacity for your platform, taking care to observe any platform specific notes and the release notes. The manual, containing a tutorial, is also accessible from the home page.

4.2 Using

On opening Audacity you will see

I. Gray, *Snake Charming—The Musical Python*,
DOI 10.1007/978-3-319-60660-6_4



and as you are only wanting to play '.wav' files press the ▷ in the top left corner. Audacity provides many more sophisticated tools for multi-track recording, analysis and playback which are outwith the scope of the current book.

Audacity® is a registered trademark of http://www.audacityteam.org.

Part II
Banging the Drum—Visualising Sound

Chapter 5
Mark Kac (1914 to 1984)

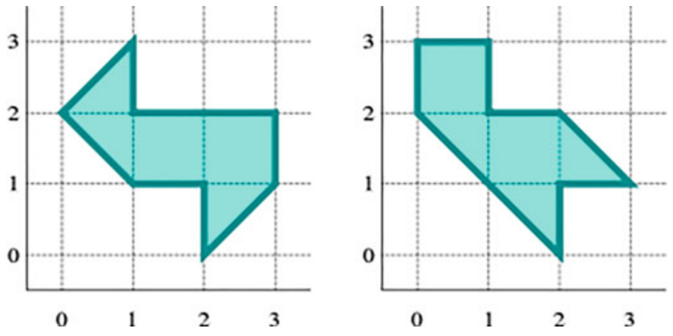
Mark Kac was a Polish mathematician specialising in probability theory, who did notable work with Pal Erdös and Richard Feynman. Today he is most often remembered for an acoustical paper on hearing drum shapes.

I. Gray, *Snake Charming—The Musical Python*,
DOI 10.1007/978-3-319-60660-6_5

5.1 Hearing the Shape of a Drum

In his seminal 1966 paper "Can you hear the shape of a drum?" Kac posed a question which was only resolved, in the negative, two and a half decades later in 1992. The paper delightfully explores the mathematics required to extract the timbre (tone colour or frequency spectrum) of an instrument with different boundary shapes (drumheads), purely from hearing it play a note. As the paper talks more about tambourines than drums, a more accurate, but slightly less catchy title, would have been "Can you hear the shape of a membranophone?"!

The question then boils down to do unique drumheads have unique timbres, or are there any cases where different shapes produce identical (isospectral) timbres? This question was immediately answered, in the negative, by the existence of a pair of differently shaped drums in the sixteenth dimension which were isospectral, but this was rather hard to visualise! However in 1992 Webb and Wolpert constructed the following pair of isospectral drums in the more familiar second dimension.

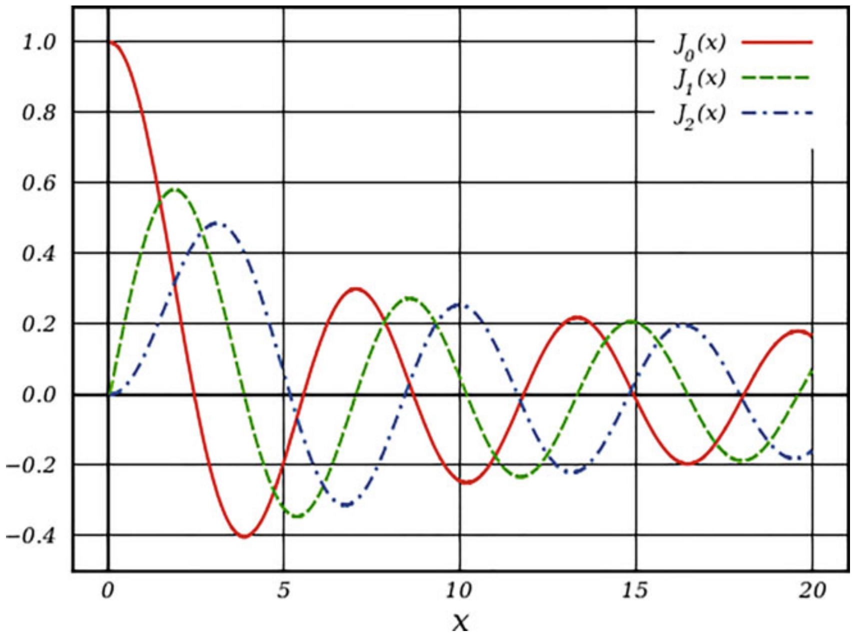


5.2 Riding the Waves—Bessel Functions

This section which is more mathematically challenging may be safely skipped, at least on first reading, without losing the theme of this chapter.

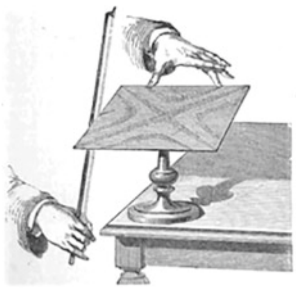
In one dimension waveforms can be described by Fourier analysis as being composed of the sum of sinusoids, at integral harmonic intervals as discussed in Chap. 7. This is true for instruments such as trumpets, violins and clarinets.

With percussion, such as drums, wave propagation over a two dimensional membrane is governed by Bessel's differential equation, $x^2 \cdot \frac{d^2y}{dx^2} + x \cdot \frac{dy}{dx} + (x^2 - \alpha^2) \cdot y = 0$. Solutions of this equation for a value of $x = 0$ are known as Bessel functions of the first kind and can be used to model the modes of vibration of a thin circular, or annular, membrane such as a tambourine or head of a drum. Bessel functions of the first kind J_α for $\alpha = 0, 1, 2$ are plotted below.

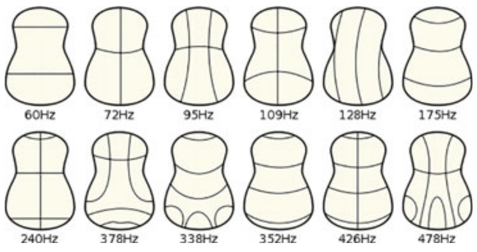


5.3 Vibrating Plates—Chladni and Germain

In the late 18th and early 19th century the German physicist Ernst Chladni, the "father of acoustics", conducted a series of experiments to determine the modes of vibration of a resonating metal plate. The experiment consisted of a centrally mounted thin square metal plate with fine sand scattered over its upper surface. This could then be bowed to resonance and fingers placed at points around the edge. The sand would settle in areas of no movement indicating the various modes of vibration of the plate.



Even today Chladni figures are still used in the design of sounding backs of stringed instruments such as a guitar.



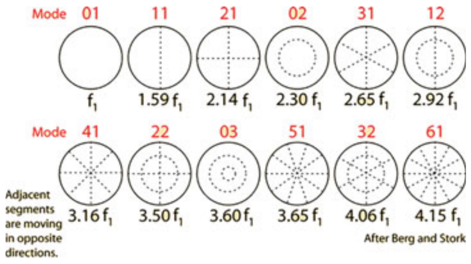
On seeing Chladni's demonstrations in Paris Napoleon offered a prize for the best mathematical explanation of the vibrating patterns observed. The prize was eventually won, at the third attempt, by Sophie Germain, a self taught mathematician: being a woman she was barred from attending the École Polytechnique or even from collecting her prize in person. Despite friendships, by correspondence, with J.L. Lagrange, A.M. Legendre and C.F. Gauss, the 'prince of mathematicians', it was only through her friendship with J. Fourier, the subject of Chap. 7, that she was able to attend sessions of the Academy. All this prejudicial treatment might provoke a latter day Cicero to exclaim "O tempores, O mores": "what times, what customs"!

Although this section has been an aside from the main theme of this chapter it has illustrated interesting parallel work being done on two dimensional elasticity.

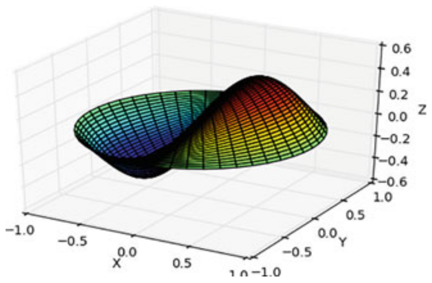
5.4 Drumhead Modes

Returning to thin elastic circular membranes which have wave propagation governed by solutions to Bessel's differential equation. The harmonics are no longer sums of integer multiples of the fundamental frequency and so euphonic (see Chap. 7). Instead drumhead modes are composed of multiple inharmonic partials, non-integral multiples of the fundamental frequency, and so are cacophonic.

In two dimensions these drumhead modes look like as contour plots


Mode 01 11 21 02 31 12
f1 1.59 f1 2.14 f1 2.30 f1 2.65 f1 2.92 f1
Mode 41 22 03 51 32 61
3.16 f1 3.50 f1 3.60 f1 3.65 f1 4.06 f1 4.15 f1
Adjacent segments are moving in opposite directions.
After Berg and Stork

Whereas in three dimensions, with height exaggeration, they look like



Animation of these modes is by a double buffering technique. Two identical Buffers

1. the Display buffer is a frozen version of the image currently being displayed
2. the Drawing buffer is a dynamic version of the next image currently being drawn

along with a pair of pointers to each buffer. On completion of the drawing the two pointers are swapped over. This makes the current Display buffer the previous Drawing buffer, and the new Drawing buffer the previous Display buffer. Compare this with a single buffer is being updated while it is being displayed, as in the Victorian Zoetrope, resulting in a much more distracting, flickery, displayed image.

Chapter 6
Project Code

Matplotlib, bundled with the Python distribution, was originally a two dimensional plotting package offering graphing facilities comparable to its commercial rivals. Recent releases have enhanced this by adding three dimensional graphics and animation capabilities. This project will develop the graphics and animation facilities required to produce an animated circular drumhead as in Chap. 14.

6.1 New Language Features

Project Headers given in New Language Features are applicable only to this section.

6.1.1 Project Header

You will need to use some features of NumPy, SciPy and Matplotlib for this project so enter here

```
import numpy as np
from scipy import special
import matplotlib.pyplot as plt
```

Where the lines mean

1. Short alias for package NumPy
2. Extract package special from SciPy
3. Short alias for package pyplot within Matplotlib.

I. Gray, *Snake Charming—The Musical Python*,
DOI 10.1007/978-3-319-60660-6_6

6.1.2 Plotting Bessel Functions

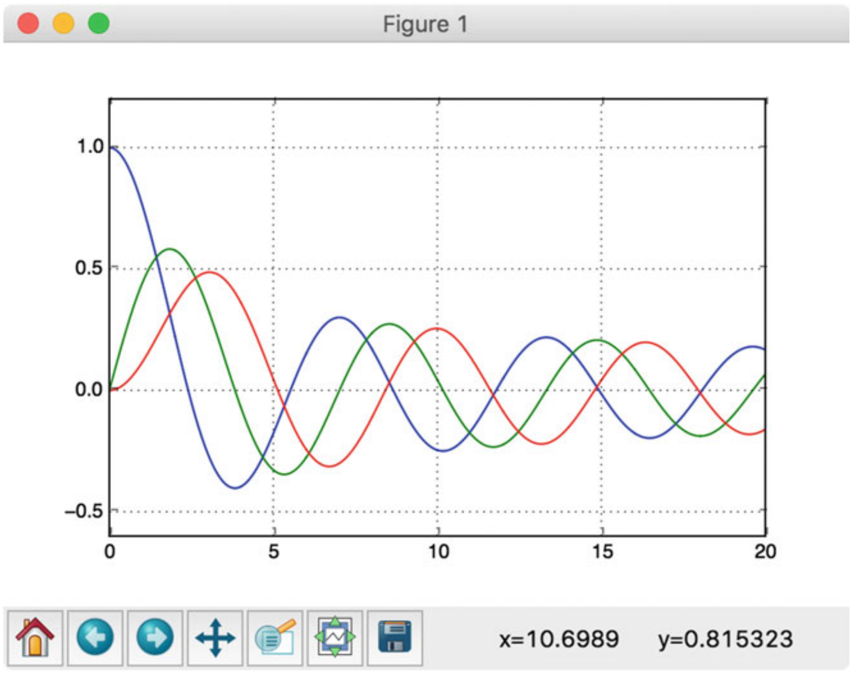
As a simple example of graphing functions type into ‘bessel.py’

```
import numpy as np
import scipy.special as spe
import matplotlib.pyplot as plt
plt.ion()
plt.grid()
x=np.linspace(0,20,1000)
for alpha in range(3):
    y=spe.jv(alpha,x)
    plt.plot(x, y, '-')
plt.show()
```

Where the highlighted lines show

1. switch on interactive plotting, after entering %matplotlib in iPython
2. plot basic grid
3. set up linearly interpolated x axis
4. set up y points as Bessel functions (first kind) for various α
5. plot (x, y) with solid line
6. show plot.

the resulting figure should be compared with that shown earlier



6.1.3 *Graphing in 3D*

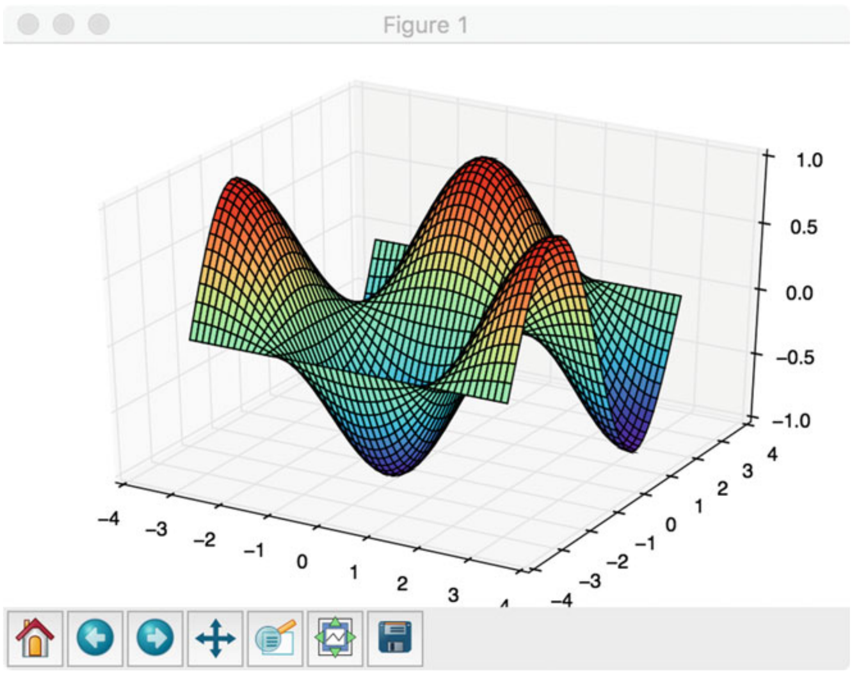
Type into 'rainbow.py'

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
plt.ion()
fig = plt.figure()
axes = Axes3D(fig)
x = np.linspace(-np.pi,np.pi)
y = np.linspace(-np.pi,np.pi)
x, y = np.meshgrid(x, y)
z=np.cos(x)*np.sin(y)
axes.plot_surface(x, y, z, rstride=1, cstride=1, cmap=plt.cm.rainbow)
plt.show()
```

Where the highlighted lines are

1. Use the `Axes3D` function within the `mplot3d` toolkit
2. setup axes
3. linearly interpolate x values
4. setup a grid over x and y values
5. plots surface with row and column spacings and using the rainbow colour map.

The resultant figure is



There is a good example of drawing a circular drumhead in the scipy.special documentation referenced in this book backmatter.

6.1.4 Animation of Square Wave from Summing Sinusoids

Although Matplotlib provides basic double buffering animation through its interactive functions

1. `ion()` allowing every plotting operation to update a figure
2. `ioff()` disabling every plotting operation from updating a figure
3. `draw()` forces a figure to be redrawn.

There is also an animation module whose most useful functions are

1. `TimedAnimation` updates the animation every time interval specified in milliseconds
2. `FuncAnimation` updates the animation every specified function call.

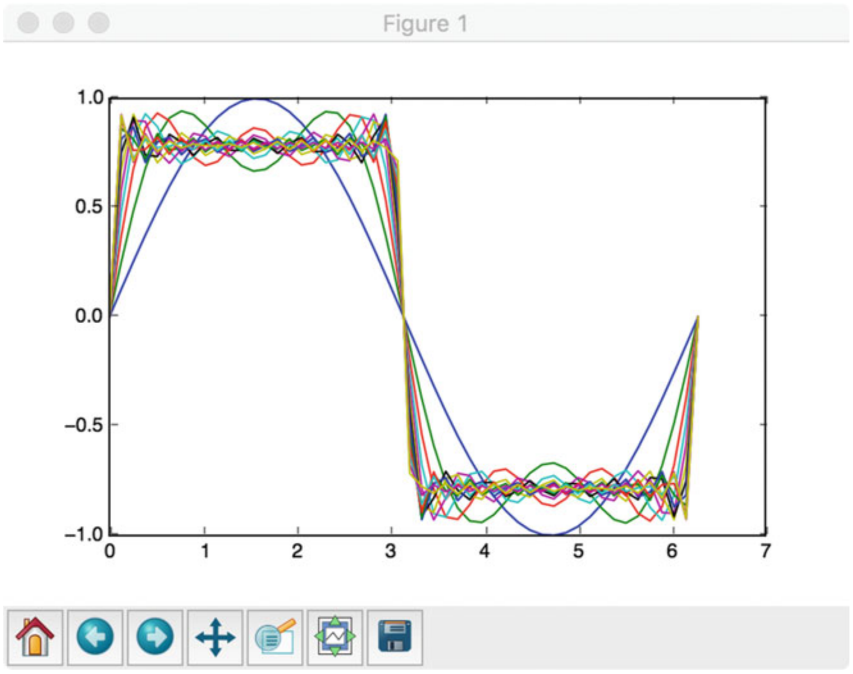
This program illustrates additive synthesis as in Chap. 8 as a simple 2D animation without redrawing. Type into 'squsin.py'

```
import numpy as np
import matplotlib.pyplot as plt
fig, ax=plt.subplots()
x=np.linspace(0,2*np.pi)
harmOdd=[1,3,5,7,9,11,13,15,17,19,21,23,25]
def synthUpdate(oddHarm):
    sumSin=0
    cnt=oddHarm
    while cnt>0:]
        sumSin=sumSin+np.sin(x*cnt)/cnt
        cnt=cnt-2
    return sumSin
for harm in harmOdd:
    y=synthUpdate(harm)
    ax.plot(x, y)
    plt.pause(1)
```

Where the highlighted lines are

1. list of the odd harmonics
2. synthesise the odd harmonics by summation
3. overdraw the figure
4. pause for one second N.B.the documentation for `plt.pause` has the following caveat "This function is experimental; its behaviour may be changed or extended in a future release." It can always be replaced by `time.delay` in this case.

The resultant figure is



6.1.5 *Animating in 3D, for Wave Propagation Along an Axis*

Using basic double buffering, clearing before redrawing a buffer, and the `mplot3d` toolkit, a very effective animation of wave propagation can be created. Type into 'propgn.py'

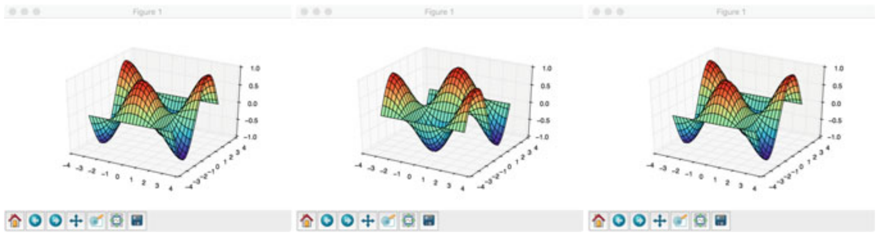
```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
x = np.linspace(-np.pi,np.pi,50)
y = np.linspace(-np.pi,np.pi,50)
x, y = np.meshgrid(x, y)
ax.set_zlim(-1, 1)
axes=None
for phi in np.linspace(-np.pi,np.pi,50):
    if axes:
        ax.collections.remove(axes)
    z=np.cos(x+phi)*np.sin(y)
```

```
        axes=ax.plot_surface(x, y, z, rstride=2,
        cstride=2,cmap=plt.cm.rainbow)
        plt.pause(0.01)
```

Where the highlighted lines show

1. Setup a 3D subplot
2. Setup z limits
3. Allow first frame to be drawn
4. Cleat buffer
5. Pause for ten milli-seconds N.B. the documentation for `plt.pause` has the following caveat "This function is experimental; its behaviour may be changed or extended in a future release." It can always be replaced by `time.delay` in this case.

Remember %matplotlib in IPython.



6.2 The Code

Although there is no separate project code for this Part the following generic header should be used for any of the extensions from Chap. 14.

6.2.1 Project Header

In 'main2.py enter

```
from math import *
from fractions import Fraction
import numpy as np
import scipy.special as spe
import matplotlib.pyplot as plt
import matplotlib.animation as anim
```

Part III
Heat in the Desert—Sculpting Sound

Chapter 7
Joseph Fourier (1768 to 1830)

Joseph Fourier (no contemporary photographs exist) a French mathematician and physicist, was recruited by Napoleon as scientific advisor to his ‘intellectual’ Army of Egypt, initially to survey a possible Suez canal. Exposure to the desert heat led to a lifelong obsession with heat and the development of Fourier Analysis.

7.1 The Army of Egypt

What on earth was Napoleon doing campaigning in Egypt and Syria between 1798 and 1801, when his normal battlefields were in Europe? The answer is simple: to threaten British trade interests with the Indian subcontinent. Of more lasting importance was his recruitment of over a hundred intellectuals to pursue scientific and

I. Gray, *Snake Charming—The Musical Python*,
DOI 10.1007/978-3-319-60660-6_7

archaeological studies in the Middle East. Amongst the discoveries of this 'intellectual army' were two that still resound today

1. the unearthing of the Rosetta stone and the subsequent decipherment of hieroglyphics by Champollion.
2. the study of heat transfer by Fourier leading to the development of Fourier Series and the Fourier Transform which are central to signal processing.

7.2 Feeling the Heat—Fourier Transforms

After being abandoned in the desert by Napoleon's return to France, Fourier eventually reached Paris in 1801. So convinced had he become of the curative effects of heat, that he kept his apartments unreasonably hot and was himself wrapped in blankets all day.

For the next six years he worked on the propagation of heat through solid bodies, developing Fourier analysis and Fourier transforms in the process. His basic tenet, that any repeating waveform could be expressed as a sum of a series of individual sinusoids, was initially received with some scepticism, even hostility, among his intellectual peers of that time. Indeed it was many years before the monumental and ground-breaking nature of his work was fully appreciated. As an aside to this work Fourier was the first to recognise and publicise the 'greenhouse effect'.

Finally because of his lifelong obsession with heat, he tripped over his blankets, fell down stairs, was put to bed and died a few days later. So be warned!

7.3 Chasing Rainbows—Frequency Spectra

Nowadays, rather than in heat propagation, the Fourier transform is used more often in audio, radio and Radar as a convenient mechanism for converting time domain signals into the frequency domain and, using the inverse transform, of converting the frequency domain into the time domain. For instance in audio it allows the extraction of the harmonic spectrum of a musical instrument playing a note.

Chapter 8
Bob Moog (1934 to 2005)

Copyright 2017, The Bob Moog Foundation

Seen here with two of his most famous instruments, the Moog Modular and the Minimoog synthesisers, Dr. Robert 'Bob' Moog was an American pioneer in the design of analogue synthesisers whose many innovations have become the industry norm of today. From the early 1950s when he designed and built Theremins, a unique instrument which you play only by moving your hands without touching it, his developments included voltage controlled oscillators, filters and amplifiers, envelope generators for controlling the musical dynamics and low frequency oscillators for modulating the sound. All of these are expanded in the section on subtractive synthesis.

© Springer International Publishing AG 2017
I. Gray, *Snake Charming—The Musical Python*,
DOI 10.1007/978-3-319-60660-6_8

8.1 Analogue Additive Synthesis

Although from the foregoing analysis in Chap. 7 it might appear that the obvious way to synthesise an instrument playing a note is to sum together its constituent sinusoids this met with several difficulties

1. percussive instruments have many inharmonic (non-integral) partials as in Chap. 5 which do not conform to standard Fourier analysis. A pealing bell, for example, has eleven partials before it begins to sound realistic.
2. timbres of certain musical instruments, particularly the brass and string sections, require a very large number of high harmonics to sound realistic.

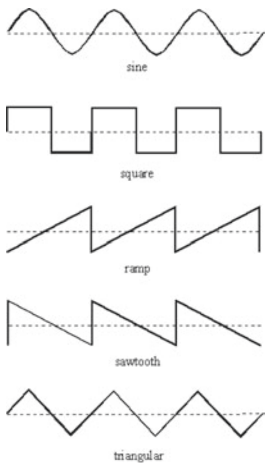
Although both of these problems can be alleviated by the use of sophisticate digital signal processing in the analogue world a different approach was required.

8.2 Analogue Subtractive Synthesis

Although the photograph at the beginning of the Chap. 8 seems a bit like an antiquated telephone exchange there are really only three main modules, two supporting modules and a keyboard to consider. A very good contemporary instrument is the Moog Sub 37

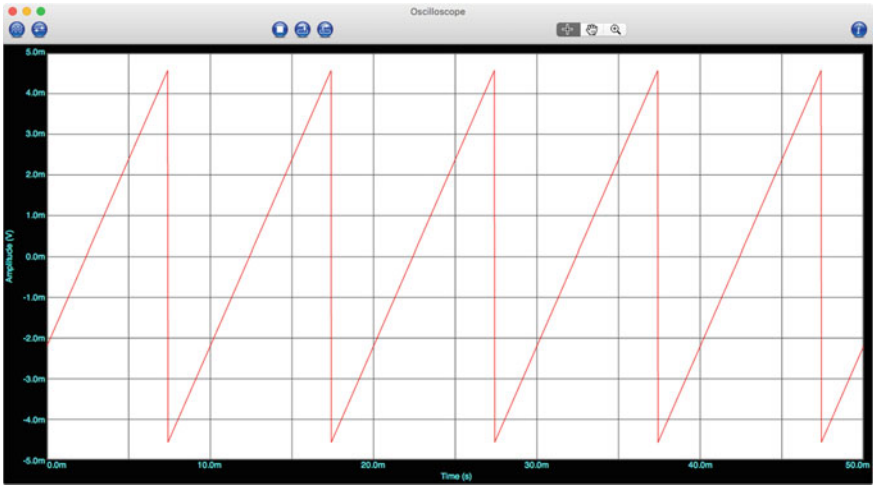
The alternative approach taken was to use a small number, usually two or three, harmonically rich waveforms, filter them to remove unwanted harmonics and finally amplify them to produce the note. An envelope generator is used to add dynamics to the output. Low frequency oscillators are used to modulate the three main modules to give effects such as vibrato and tremolo. Finally a keyboard supplies the frequency and duration of the note played.

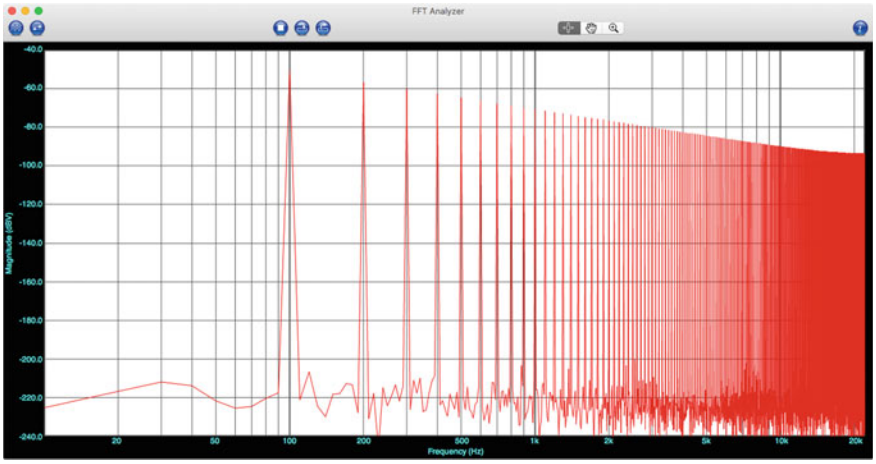
8.2.1 Oscillators



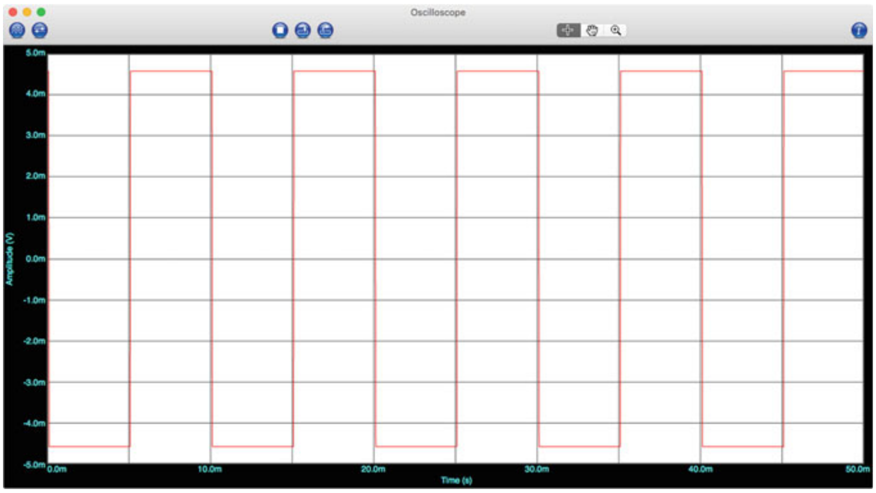
Only three of the waveforms depicted here, the sawtooth, square and triangle, are used for oscillator output. All five are used for the low frequency oscillator in the Modulation section. The input to an oscillator is a frequency in Hertz (Hz) which is the number of complete repetitions of the waveform in a second. All the waveforms and their spectra depicted below have a fundamental frequency of 100 Hz, and were performed at a sample rate of 44.1 kHz.

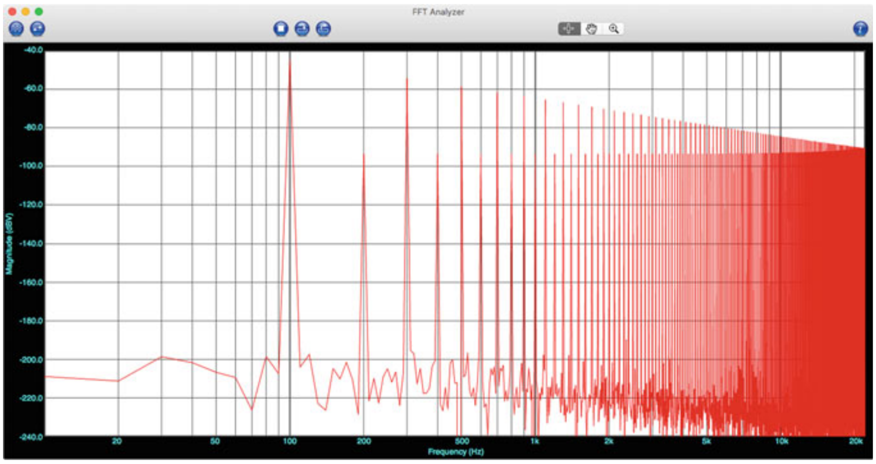
The figures were all produced using Faber Acoustical's SignalSuite and SignalScope software.



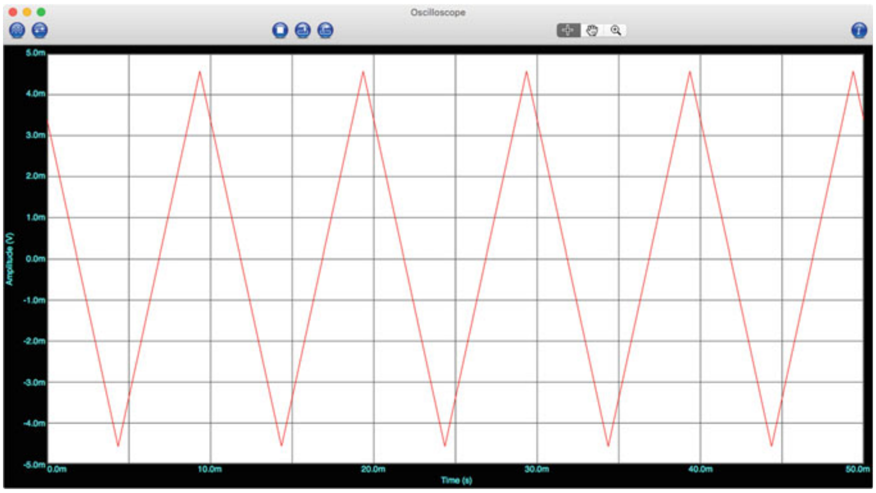


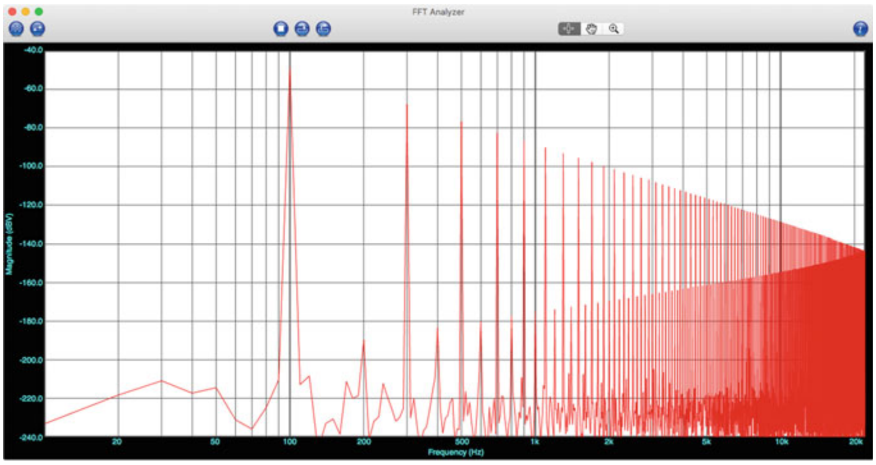
Sawtooth oscillators have a full set of harmonics and are are particularly good at forming the basis for the bright sounds of brasses or bowed strings.





Square wave oscillators have only odd numbered harmonics and are ideal as the basis for clarinets. For oboes and beyond pulse width modulation must be performed to transform the square into an increasingly rectangular waveform.





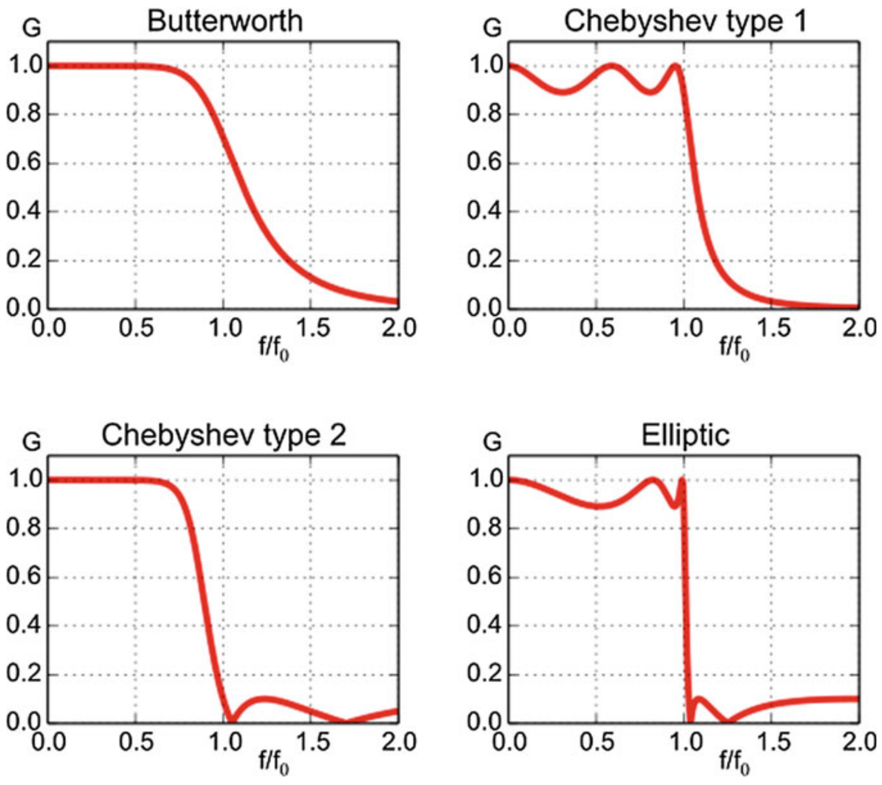
Triangle wave oscillators again have only odd numbered harmonics but these decay at a far faster rate than they do in the square wave. They are most suited for flute and thin reed sounds.

Notes to observant, and hopefully still interested, readers

1. the first waveform was a ramp rather than a sawtooth - mea culpa! However their spectra are identical as there is only a phase-shift of 180 degrees between the waveforms.
2. the spectral folding/aliasing at 22 kHz is an artefact of the Shannon/Nyquist Sampling Theorem, and does not invalidate the spectral content being all, or odd numbered, harmonics.

8.2.2 *Filters*

A filter has a dramatic effect on the timbre of a note. It is most often a low pass filter which attenuates frequencies sharply above its cutoff frequency, while leaving those below unchanged. This affects whether the note sounds warm and dark or cold and bright.



The characteristics of the filters illustrated are

1. large attenuation of signals above the cutoff frequency
2. resonance or emphasis of signals about the cutoff frequency
3. filter slope (Q) indicating the rate of attenuation usually expressed in -dBV/octave

Note that the cutoff frequency is usually taken as the point when -3dBV attenuation below the main flat response has occurred.

Butterworth filters will be used in the project code, with Chebyshev 1 used to introduce resonance as in Chap. 14.

Other types of filters are used less commonly including high pass, band pass and band stop filters which ‘do exactly as they say on the tin’. All pass filters do not attenuate frequencies but cause a phase shift between input and output.

Filters are further classified by whether they involve feedback of the output into the input as

1. Finite Impulse Response (FIR) filters have no feedback
2. Infinite Impulse Response (IIR) filters involve feedback

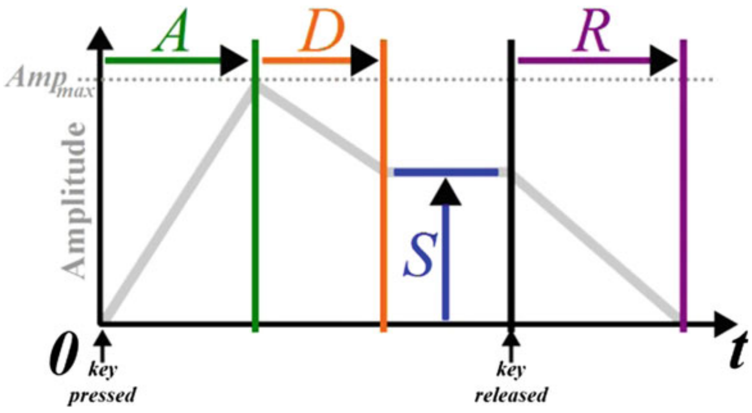
Most of the filters used here will be of the IIR type.

8.2.3 Amplifier

The amplifier controls the final output level of the sound which can be affected by the envelope generators and modulators.

8.2.4 Envelope Generation

The overall dynamics, how the sound builds up and dies away, is controlled by an amplifier envelope generator.



It traditionally has the four stages shown above

1. Attack phase—the time it takes to reach a maximum output level from the key being pressed
2. Decay phase—the time it takes to fall to a sustain level
3. Sustain level—the volume the note is held at until the key is released
4. Release phase—the time to reach zero output level after the key is released

and is known as an amplitude ADSR envelope generator under the control of a key on/off gate.

8.2.5 Modulation

Modulation uses a low frequency oscillator in the infra-sonic range (0–30 Hz) to modify waveforms in the three core modules. Any waveform in the oscillators section can be used. For example using a sine waveform and targeting the

1. oscillator gives frequency modulation (FM)—vibrato
2. amplifier gives amplitude modulation (AM)—tremolo

Chapter 9
Project Code

This project will develop the core modules of an analogue subtractive synthesiser, with audio monitoring of each module and enable you to

1. change frequency and waveforms of the oscillators, and their mixtures
2. change the cutoff frequency and filter slope of the low pass filter
3. change the output volume of the amplifier
4. change the attack, decay, sustain and release of the amplifier envelope generator
5. change the rate and amount of the low frequency modulation

Further time domain and frequency domain displays will be developed, using Digital Signal Processing, to show respectively the waveform and spectral content of the output from the oscillator, filter and amplifier.

9.1 New Language Features

Project Headers given in New Language Features are applicable only to this section.

9.1.1 Using Tkinter

The graphical user interface (GUI) of the synthesiser will be built from these widgets within the 'tkinter' toolbox which is common across all platform. Insert the following into the program header

I. Gray, *Snake Charming—The Musical Python*,
DOI 10.1007/978-3-319-60660-6_9

```
from tkinter import *
from tkinter import ttk
from tkinter import colorchooser
root=Tk(()
```

Where these four lines establish

1. tkinter as main package
2. ttk (themed Tk) as the widget package available in Tcl/Tk 8.5 onward
3. colorchooser package
4. root as Tk root level

Within the root level GUI there will be frames supporting

1. the oscillators' frequency and amount of pulse width modulation will be determined by list menus, and the mixture between them by mouse position within a system colour chooser graphical element
2. the filter's cutoff and slope will be list menu items
3. overall volume will be controlled by a slider
4. ADSR settings on the amplitude envelope generator will be by sliders
5. modulation rate and amount will be set by list menus, and its destination by a radio box (off, vibrato or tremolo)
6. displays for the oscillators', filter's and amplifier's outputs will be controlled by on/off check boxes

9.1.2 Project Header

You will need to use some features of NumPy, SciPy and Matplotlib for this project so enter here

```
import numpy as np
import matplotlib.pyplot as plt
```

9.1.3 Sound Storage

Arrays from NumPy will be used to hold the numerical samples of the sound. These will be turned into WAV files for input, output and offline processing of these sounds. Add the following line to the project header

```
from scipy import io
```

9.1.4 Harmonic Analysis

The manipulation and display of the harmonic content of sounds requires the use of the Fast Fourier Transform (FFT). Add the following line to the project header

```
from scipy import fft
```

9.1.5 Oscillators and Mixer

This section is long and repetitive because it provides the basis for sound generation both here and in Chap. 13.

Insert the following Python test program saving it as ‘waves.py’

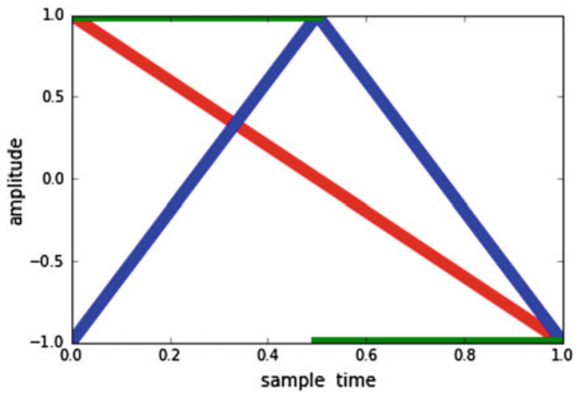
```
# oscillators for mixing
import matplotlib.pyplot as plt
plt.plot([0,1],[1,-1],'r-',[0,.5,1],[-1,1,-1],'b-',
            [0,.5],[1,1],'g-',[.5,1],[-1,-1],'g-',linewidth=8)
plt.xlabel('sample time')
plt.ylabel('amplitude')
plt.show()
# samples of 1 second duration
from numpy import linspace, append
from scipy.io.wavfile import read, write
freqSamp=44100
freqNote=210
numRpts=freqSamp//freqNote
print(freqNote,numRpts)
test=linspace(1.0,-1.0,16,False).astype(float)
print(test)
test2=append(test,test)
print(test2)
sawData=linspace(1.0,-1.0,numRpts,False).astype(float)
rpts=freqNote-1
sawNote=sawData
while rpts>0:
    sawNote=append(sawNote,sawData)
    rpts=rpts-1
print('sawtooth',sawNote.shape )
write('sawtooth.wav',freqSamp,sawNote)
halfRpts=numRpts//2
firstHalf=linspace(-1.0,1.0,halfRpts,False).astype(float)
lastHalf=linspace(1.0,-1.0,halfRpts,False).astype(float)
```

```
triData=append(firstHalf,lastHalf)
rpts=freqNote-1
triNote=triData
while rpts>0:
    triNote=append(triNote,triData)
    rpts=rpts-1
print('triangle',triNote.shape)
write('triangle.wav',freqSamp,triNote)
firstHalf=linspace(1.0,1.0,halfRpts,False).astype(float)
lastHalf=linspace(-1.0,-1.0,halfRpts,False).astype(float)
squData=append(firstHalf,lastHalf)
rpts=freqNote-1
squNote=squData
while rpts>0:
    squNote=append(squNote,squData)
    rpts=rpts-1
print('square',squNote.shape)
write('square.wav',freqSamp,squNote)
# mixing in pairs
sawtriNote=(sawNote+triNote)/2
print('saw and tri')
write('sawtri.wav',freqSamp,sawtriNote)
trisquNote=(triNote+squNote)/2
print('tri and squ')
write('trisqu.wav',freqSamp,trisquNote)
squsawNote=(squNote+sawNote)/2
print('squ and saw')
write('squsaw.wav',freqSamp,squsawNote)
# mixing all three waveforms
sawtrisquNote=(sawNote+triNote+squNote)/3
print('all three')
write('sawtrisqu.wav',freqSamp,sawtrisquNote)
# reading and mixing
(rate,saw)=read('sawtooth.wav')
(rate,squ)=read('square.wav')
(rate,tri)=read('triangle.wav')
print(rate,saw)
print(rate,squ)
print(rate,tri)
print('saw3squ2tri1')
mixNote=saw/2+squ/3+tri/6
write('saw3squ2tri1.wav',rate,mixNote)
```

whose output is a single wave of the sawtooth(red), triangle(blue) and square(green) oscillators for repeated storage in their respective '.wav' files, and then output for one second, then mixtures of two or three of these waveforms and finally reading back the waveforms and mixing them in proportions.

The code is in five sections each headed by a highlighted comment.

The first section gives an example of a few pyplot commands to draw the oscillators' waveforms. `plt.plot` is followed by two arrays of x and y coordinates for the line. If only one array is given it is assumed to be the y coordinates. The string following gives the line's colour and appearance and is optionally followed by parameters describing for instance the line width. Because of the double buffering technique described in Chap. 5 the output will only be displayed after the `plt.show()` function call. This displays a single waveform of each of the three oscillators.



The second section constructs these waveforms repeating them `freqNote` times and writing their output to '.wav' files. There is no reason why the sampling frequency `freqSamp` needs to be the same as the playback frequency but doing so avoids the need for messy interpolation routines. The CD sampling frequency of 44100 Hz is used, giving a Nyquist (Chap. 8) frequency of 22050 Hz just above the human hearing range. $44100 = (2 \cdot 3 \cdot 5 \cdot 7)^2$ has 79 factors (see http://factornumber.com/?page=44100) enabling a choice of integer factorisations for frequency with an even number of repeats to allow construction of square and triangle waveforms. These frequencies are

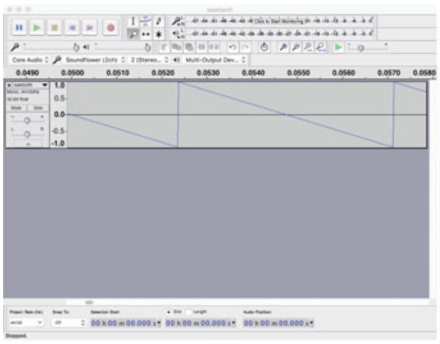
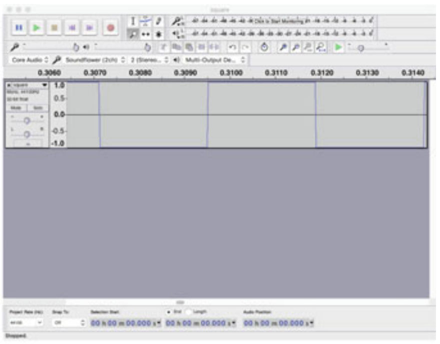
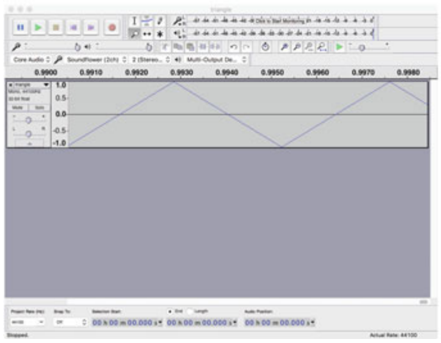
```
allowFreqs=[22050,11025,7350,4410,3675,3150,2450,2205,
            1575,1470,1225,1050,882,735,630,525,490,
            450,441,350,315,294,245,225,210,180,175,
            150,147,126,105,98,90,75,70,63,60,50,45,42,35,30]
```

thus covering the human range of frequencies of 30 Hz $\rightarrow$ 20 kHz. `linspace` performs linear interpolation over a range for a number of points, where False says not to include the endpoint. The variable `test1` illustrates the operation. `append` joins two arrays to form a longer array. The variable `test2` illustrates the operation. The waveform is built by first forming a single copy using `numRpts` and appending `freqNote` copies to make an one second note. For square and triangle waveforms the first stage must be performed as two half waveforms appended together. Finally `write` puts these out to a '.wav' file, at a rate of `freqSamp` using the constructed waveform data. There is obviously a lot of repetition here which will be culled out in the final code but as the Bellman says

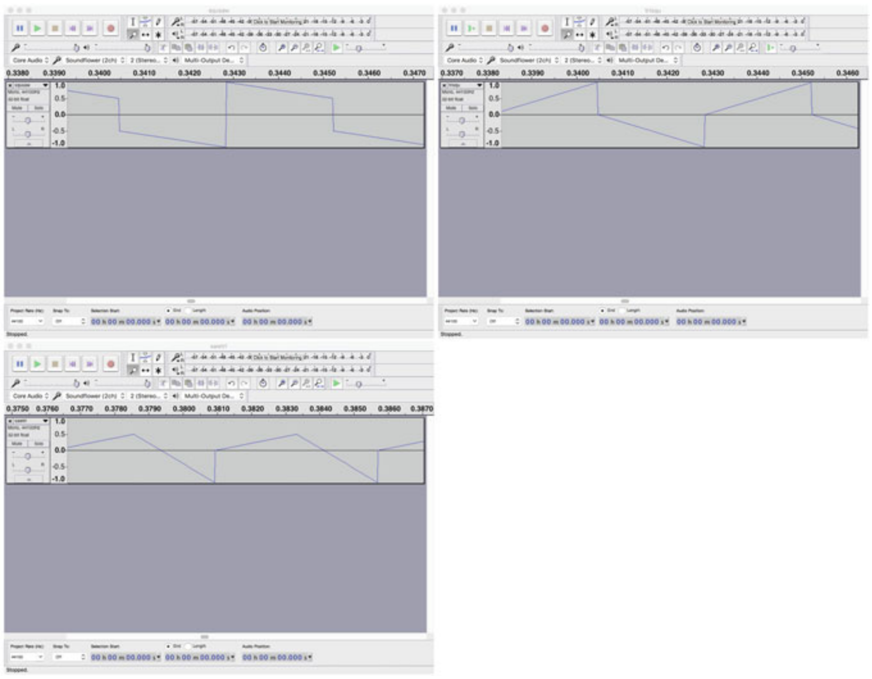
"what I tell you three times is true"

Lewis Carroll, 'The Hunting of the Snark'

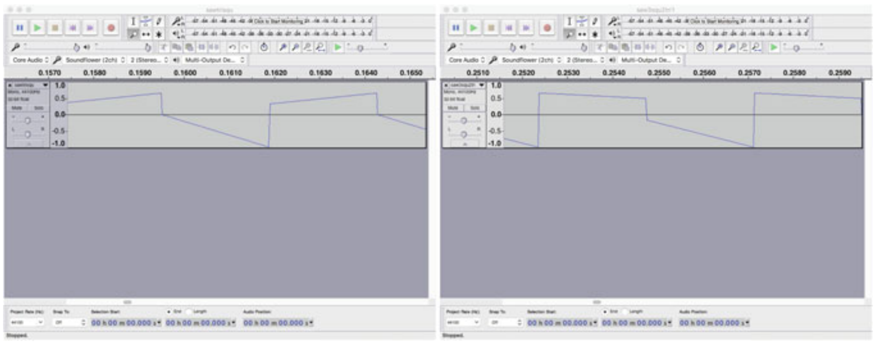
Audacity displays of sawtooth, square and triangle waveforms are illustrated below.

The third section shows how to mix waveforms in pairs and normalise them before writing them out. Audacity displays of mixtures of sawtooth and square, square and triangle and sawtooth and triangle waveforms are illustrated below.



The fourth and fifth sections show an equal and a proportional mix of all three waveforms and also how to `read` back a '.wav' file. Audacity displays of an equal and proportional mix are illustrated below, and demonstrate the variety of timbres achievable purely by mixing alone.



9.1.6 Low Pass Filtering

Filters are used to remove unwanted parts of a signal such as high harmonics leaving the rest of the signal untouched. A filter taking an average over a series of time steps provides the basis of a low pass filter. This can be of two forms, where x(n) and y(n) are the input and output at time n respectively.

1. Finite Impulse Response (FIR) where the output is purely a function of the input. A simple example is a moving window average filter such as $y(n) = \frac{1}{5} \cdot \sum_{i=0}^{4} x(n-i)$
2. Infinite Impulse Response (IIR) is a function of the input and feedback of previous values of the output. A simple example is an alpha filter $y(n) = \alpha \cdot x(n) + (1-\alpha) \cdot y(n-1)$

Insert the following Python test program saving it as ‘alpha.py’

```
# reading samples of 1 second duration
from numpy import linspace
from scipy.io.wavfile import read, write
(freqSamp,sawwav)=read('sawtooth.wav')
(freqSamp,squwav)=read('square.wav')
(freqSamp,triwav)=read('triangle.wav')
# initialising filter
sawflt=linspace(0.0,0.0,freqSamp,False).astype(float)
squflt=linspace(0.0,0.0,freqSamp,False).astype(float)
triflt=linspace(0.0,0.0,freqSamp,False).astype(float)
alpha=1/20
sawflt[0]=alpha*sawwav[0]
squflt[0]=alpha*squwav[0]
triflt[0]=alpha*triwav[0]
# alpha filter
for flt in range(1,freqSamp):
    sawflt[flt]=alpha*sawwav[flt]+(1-alpha)*sawflt[flt-1]
    squflt[flt]=alpha*squwav[flt]+(1-alpha)*squflt[flt-1]
    triflt[flt]=alpha*triwav[flt]+(1-alpha)*triflt[flt-1]
# writing filtered samples
write('sawflt.wav',freqSamp,sawflt)
write('squflt.wav',freqSamp,squflt)
write('triflt.wav',freqSamp,triflt)
print('saw',sawwav)
print('flt',sawflt)
print('squ',squwav)
print('flt',squflt)
print('tri',triwav)
print('flt',triflt)
```

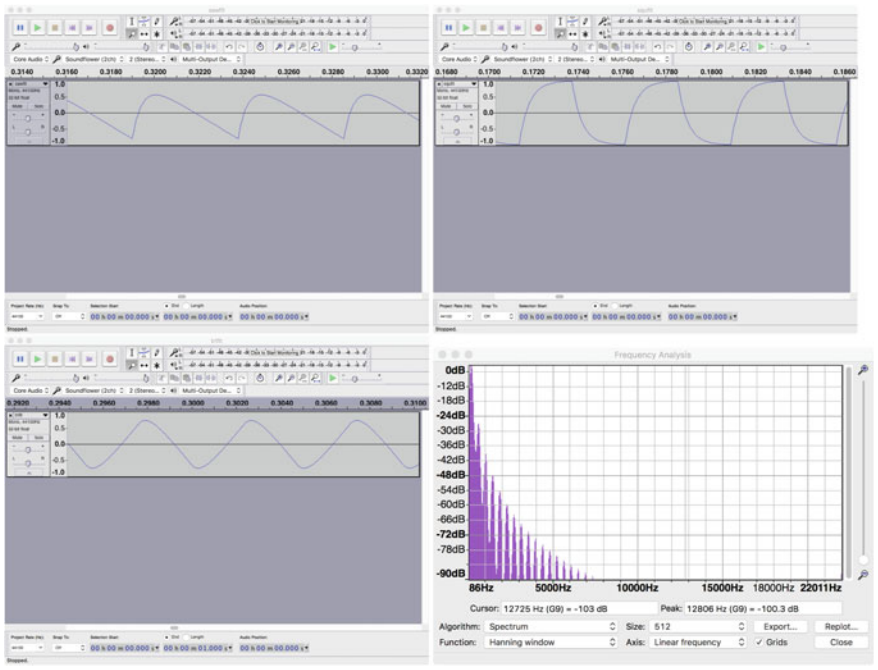
The code is in four sections each headed by a highlighted comment.

The first section reads in unmixed copies of the original three oscillators.

The second section allocates array space for the filtered outputs, sets alpha for the filter and sets the initial output by assuming y(-1)=0.

The third section is the alpha filter, $y(n) = \alpha \cdot x(n) + (1-\alpha) \cdot y(n-1)$ itself.

The final section writes out the filtered values printing them for comparison purposes. Although the alpha filter is a blunt instrument for performing keyhole surgery, it has ‘rounded off’ the sharp corners in the sawtooth, square and triangle waveforms shown below. The filter cutoff is about 6000 Hz as shown in the triangle spectrum. This combined effect gives a sound which is mellower, warmer and darker than the original. More refined filters within the package `scipy.signal` will be used in the final code.



Cursor: 12725 Hz (G9) = -103 dB
Peak: 12806 Hz (G9) = -100.3 dB
Algorithm: Spectrum
Size: 512
Function: Hanning window
Axis: Linear frequency
✓ Grids

9.1.7 *Implement Butterworth Low Pass Filter*

As an example of the finer control possible implement a twelfth order Butterworth filter with a cutoff of 5000 Hz, saving it as ‘butter.py’

```
# calculate Butterworth Low-pass coordinates
import numpy as np
from numpy import zeros
from scipy.io.wavfile import read, write
from scipy.signal import butter, lfilter, lfilter_zi
(freqSamp,triwav)=read('triangle.wav')
# initialising filter
tribut=zeros(freqSamp)
order=12
cutoff=5000
Nyquist=freqSamp/2
```

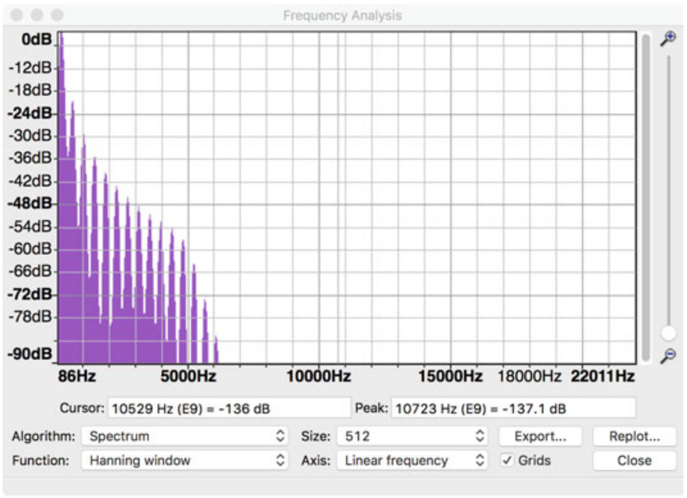
```
Wn=cutoff/Nyquist
(b,a)=butter(order,Wn)
zi=lfilter_zi(b,a)
# applying filter
(tribut,_)=lfilter(b,a,triwav,zi=zi*triwav[0])
# writing filtered samples
write('tribut.wav',freqSamp,tribut)
```

Where the five highlights are

1. the Nyquist frequency is half the sampling rate, see Chap. 8
2. for a digital filter the normalisation is [0, 1] where 1 is the Nyquist frequency
3. the Butterworth design returns a tuple of the numerator (b) and denominator (a) polynomials of the IIR filter, see Chap. 8
4. computes the initial state for the filter
5. executes the fitter for the given design and initial conditions

the last three stages need to be performed for each new filter type such as `scipy.signal.cheby1`.

The Audacity spectrum for this filter exhibits a sharp 'shoulder' at the cutoff and should be compared with that from the alpha filter above.



Cursor: 10529 Hz (E9) = -136 dB
Peak: 10723 Hz (E9) = -137.1 dB
Algorithm: Spectrum
Size: 512
Function: Hanning window
Axis: Linear frequency
✓ Grids

9.1.8 Amplitude Envelope Generation

The dynamics of an instrument's sound over time are controlled by the amplitude envelope generator.

Insert the following Python test program saving it as ‘ADSR.py’

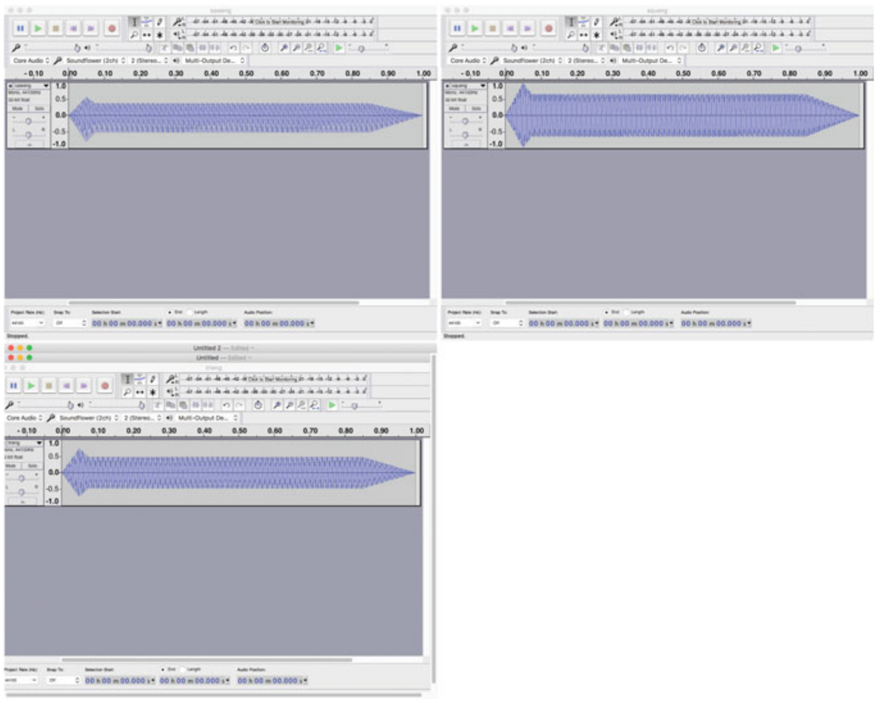
```
# reading samples of 1 second duration
from math import *
from numpy import linspace,append
from scipy.io.wavfile import read, write
(freqSamp,sawwav)=read('sawflt.wav')
(freqSamp,squwav)=read('squflt.wav')
(freqSamp,triwav)=read('triflt.wav')
onems=freqSamp/1000
# ADSR envelope
attack=floor(50 * onems)
decay=floor(25 * onems)
release=floor(150 * onems)
sustain=0.65
susDur=freqSamp-(attack+decay+release)
# 4 linspace for envelope
A=linspace(0,1,attack,False).astype(float)
D=linspace(1,sustain,decay,False).astype(float)
S=linspace(sustain,sustain,susDur,False).astype(float)
R=linspace(sustain,0,release,False).astype(float)
AD=append(A,D)
ADS=append(AD,S)
ADSR=append(ADS,R )
print(ADSR,ADSR.shape,type(ADSR))
print(sawwav,sawwav.shape,type(sawwav))
saweng=ADSR*sawwav
print(saweng,saweng.shape,type(saweng))
squeng=ADSR*squwav
trieng=ADSR*triwav
write('saweng.wav',freqSamp,saweng)
write('squeng.wav',freqSamp,squeng)
write('trieng.wav',freqSamp,trieng)
```

The code is in three sections each headed by a highlighted comment.

The first section reads in the filtered versions of the unmixed oscillators, and sets a time standard of one milli-second.

The second section sets the durations of each envelope component and the output sustained level.

The third section uses `linspace` to interpolate the four parts of the envelope which are then joined using `append`. This envelope is then **multiplied** by the waveform to account for both positive and negative segments. The output files are shown below for filtered sawtooth, square and triangle waveforms.



9.1.9 Low Frequency Oscillator and Modulation

A Low Frequency Oscillator (LFO) operates at infra-sonic frequencies (0 Hz → 30 Hz) to produce a signal to modulate one of the three main modules, the Voltage Control Oscillator, Filter and Amplifier (VCO, VCF and VCA). With a frequency sampling rate of 44100 Hz any of the following frequencies in Hz are allowed

```
allowMods=[30,25,21,18,15,14,10,9,7,6,5,3,2,1]
```

Insert the following Python test program saving it as ’sinemod.py’

```
# generate 5Hz sine wave
from math import *
from numpy import append,empty
from scipy.io.wavfile import read, write
freqSamp=44100
freqNote=5
sampInt=freqSamp//freqNote
sampSine=empty(sampInt).astype(float)
print(sampSine,sampSine.shape,type(sampSine))
twopi=2*pi
sampSine[0]=0.0
```

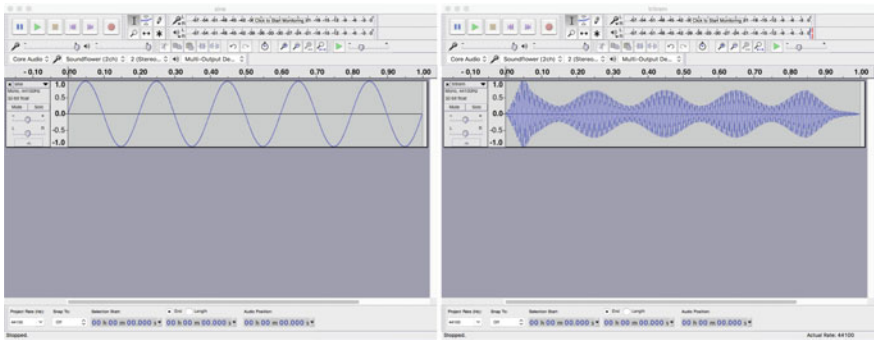
```
time=1
while time<sampInt:
    sampSine[time]=sin(twopi*time/sampInt)
    time=time+1
print(sampSine)
sineWav=sampSine
cnt=freqNote-1
while cnt>0:
    sineWav=append(sineWav,sampSine)
    cnt=cnt-1
write('sine.wav',freqSamp,sineWav)
amount=0.5
mod=sineWav*amount
# tremolo
(rate,trieng)=read('trieng.wav')
write('tritrem.wav',rate,trieng*(1+mod))
```

The code is in two sections each headed by a highlighted comment.

The first section proceeds as before for assembling waveforms, except that `linspace` is not used and the sine wave is interpolated manually. Other modulating waveforms in Chap. 8, sawtooth, ramp, square and triangle, can all be made with `linspace` as above.

The second section applies Amplitude Modulation (AM) to the VCA to produce tremolo. Frequency Modulation (FM) on the VCO will be discussed in Chap. 14.

The illustrations are of the 5 Hz sine wave and tremolo applied to the filtered and envelope triangle wave.



9.1.10 Analysis Displays

The analysis displays will allow showing a single waveform (in the time domain), and its harmonic, spectral content (in the frequency domain).

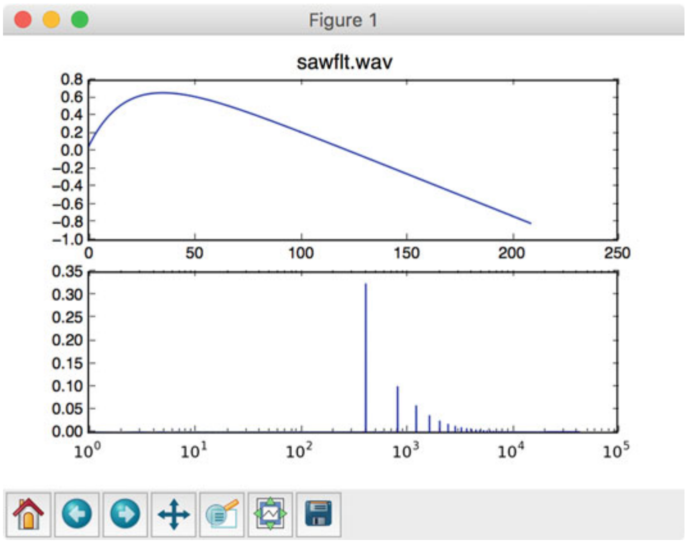
Insert the following Python test program saving it as 'display.py'.

```
# time and frequency displays
from math import *
import numpy as np
from scipy.io.wavfile import read
from scipy.fftpack import rfft
import matplotlib.pyplot as plt
fileName='sawflt.wav'
(freqSamp,wavData)=read(fileName)
freqNote=210
print(freqSamp,wavData)
plt.ion()
plt.figure(1)
plt.suptitle(fileName,fontsize=18)
plt.subplot(211)
plt.plot(wavData[0:freqNote])
plt.subplot(212)
plt.semilogx()
plt.plot(abs(rfft(wavData))/32768)
```

The code is straightforward but several important new concepts are highlighted. These are

1. `scipy.fftpack` contains functions which perform the Discrete Fourier Transform from the time domain to the frequency domain efficiently. This transformation uses Fast Fourier Transforms (FFT).
2. `plt.ion` sets up interactive graphics on typing `%matplotlib` in the IPython window.
3. the next three lines set up multiple subplots within a figure.
4. the next line draws a single waveform using an array slice.
5. the final two lines produce the frequency display using a real FFT scaled to [0,1].

The final display, with interactive controls for panning and zooming, looks like



9.1.11 *Graphical User Interface*

The widgets from tkinter that will form the GUI will be

Overall	six **Frame**s to contain each major unit in synthesiser
Overall	a **Button** to start the program when all user entries, including default values, are completed
Oscillator frame	containing a **Combobox** to allow selection of frequencies from a list and a mixer **Button** calling up a **colorchooser** to mix the three different oscillators
Filter frame	two **Radiobutton**s will allow selection of the filter's order and cutoff frequency
Amplifier frame	a **Radiobutton** to select output volume level
Envelope Generator frame	four **Entry** boxes allowing the input of ADSR values in msec and %
Display and Storage frame	a **Radiobutton** allowing a choice of VCO, VCF or VCA outputs to be displayed, and an **Entry** for specifying sound storage file
LFO frame	a **Combobox** to choose the modulation frequency and a **Radiobutton** to select its depth

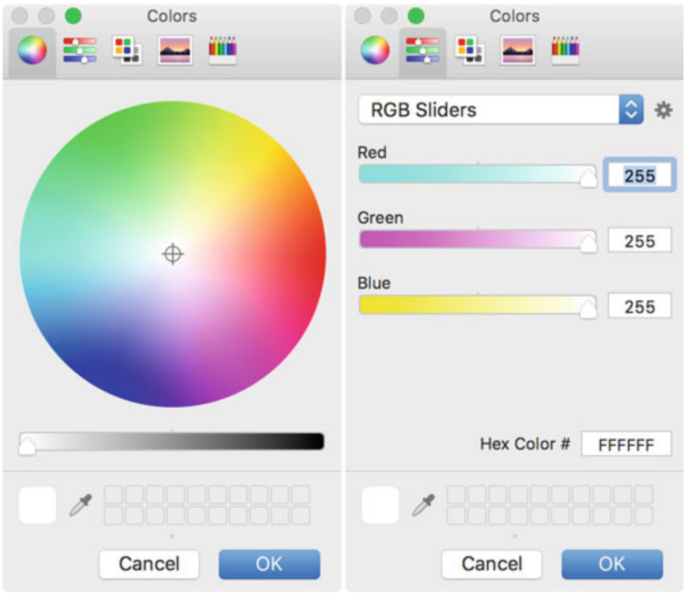
9.1.12 GUI Support for Mixer Button

The Mixer will be called from a **Button** within the VCO **Frame**. Type into 'mixer.py'

```
from tkinter import *
from tkinter import ttk
from tkinter.colorchooser import *
def getColor():
    ((red, green,blue), color) = askcolor()
    print('sawtooth = ', red)
    print('square = ', green)
    print('triangle = ', blue)
    print('color = ', color)
    return color
root=Tk()
ttk.Button(root, text='Mixer', command=getColor).
pack()
root.mainloop()
```

where colour is handled as a nested tuple. Note that the program is contained in a `root.mainloop()` GUI event loop it must be stopped by quitting Spyder.

The mixer **Button** will call up this palette for mixing sawtooth(red), triangle(blue) and square(green) oscillators in proportions. The displays are illustrated from the Macintosh.



Red 255
Green 255
Blue 255
Hex Color # FFFFFF

9.2 The Code

The project Python code is thus a composite of the above subsections, with suitable modifications to accommodate the GUI. Individual sections will be encapsulated in functions, with sound file data being the parameter mainly used between them. It develops the three major functional units, of the synthesiser, VCO, VCF and VCA, and two supplementary units AEG and LFO, along with displays and sound storage functions.

9.2.1 Project Header

Type into ‘main3.py’

```
from math import *
import numpy as np
import scipy.io.wavfile as io
import scipy.signal as sig
import scipy.fftpack as fft
import matplotlib.pyplot as plt
from tkinter import *
from tkinter import ttk
import tkinter.colorchooser as col
freqSamp = 44100
```

9.2.2 Oscillator and Mixer

Within the VCO the oscillators , sawtooth, square and triangle, provide the sound generation functions, which the mixer the mixer combines into a single piece of sound data for further processing.

Inputs	frequency of note, oscillator mixture
Output	sound data for mixed oscillators

Type into ‘main3.py’

```
# oscillators and mixer
def VCO(freqNote=441,mix=0x7F7F7F):
    def oscillators():
        def Saw():
            numRpts=freqSamp//freqNote
            sawData=np.linspace(1.0,-1.0,numRpts,False).astype(float)
            rpts=freqNote-1
            sawNote=sawData
            while rpts>0:
                sawNote=np.append(sawNote,sawData)
                rpts=rpts-1
            return sawNote
        def SquTri():
            halfRpts=freqSamp//(2*freqNote)
            firstHalf=np.linspace(1.0,1.0,halfRpts,False).astype(float)
            lastHalf=np.linspace(-1.0,-1.0,halfRpts,False).astype(float)
            squData=np.append(firstHalf,lastHalf)
            firstHalf=np.linspace(-1.0,1.0,halfRpts,False).astype(float))
            lastHalf=np.linspace(1.0,-1.0,halfRpts,False).astype(float)
            triData=np.append(firstHalf,lastHalf)
            rpts=freqNote-1
            squNote=squData
            triNote=triData
            while rpts>0:
                squNote=np.append(squNote,squData)
                triNote=np.append(triNote,triData)
                rpts=rpts-1
            return (squNote,triNote)
        sawNote=Saw()
        (squNote,triNote)=SquTri()
        return (sawNote,squNote,triNote)
    def mixer(oscs):
        (saw,squ,tri)=oscs
        blue=mix%256
        green=(mix//256)%256
        red=(mix//65536)
        norm=red+green+blue
        return (red*saw+green*squ+blue*tri)/norm
    return mixer(oscillators())
```

9.2.3 Filter

The VCF is the principal sound shaper within the synthesiser attenuating frequencies above the cutoff at a rate dependent on the order of the filter.

Inputs	sound data, cutoff frequency and order of Butterworth filter
Output	filtered sound data

Type into ‘main3.py’

```
# filter
def VCF(soundData,cutoff=5000,order=4):
# initialising filter
    buffer=np.zeros(freqSamp)
    Nyquist=freqSamp/2
    Wn=cutoff/Nyquist
    (b,a)=sig.butter(order,Wn)
    zi=sig.lfilter_zi(b,a)
# applying filter
    (buffer,_)=sig.lfilter(b,a,soundData,zi=zi*soundData[0])
    return buffer
```

9.2.4 Amplitude Envelope Generator

The AEG controls the overall dynamics of the synthesiser sound dependent on an envelope.

Inputs	sound data, values for Attack, Decay, Sustain and Release
Output	dynamically shaped sound data

Type into ‘main3.py’

```
# amplitude envelope generator
def AEG(soundData,envelope=(10,10,0.5,10)):
    onems=freqSamp/1000
# ADSR envelope
    (att,dec,sus,rel)=envelope
    attack=floor(att * onems)
    decay=floor(dec * onems)
    sustain=sus
    release=floor(rel * onems)
    susDur=freqSamp-(attack+decay+release)
# 4 linspace for envelope
    A=np.linspace(0,1,attack,False).astype(float)
    D=np.linspace(1,sustain,decay,False).astype(float)
    S=np.linspace(sustain,sustain,susDur,False).astype(float)
    R=np.linspace(sustain,0,release,False).astype(float)
    AD=np.append(A,D)
    ADS=np.append(AD,S)
    ADSR=np.append(ADS,R)
    return ADSR*soundData
```

9.2.5 Modulator

The Low Frequency Oscillator (LFO) allows modulation of the sound by a low frequency waveform.

Inputs	sound data, LFO frequency and depth
Output	modulated sound data

Type into ‘main3.py’

```
# LFO modulation
def LFO (soundData,LFOrate=5,depth=0.5):
    sampInt=freqSamp//LFOrate
    sampSine=np.empty(sampInt).astype(float)
    twopi=2*np.pi
    sampSine[0]=0.0
    time=1
    while time<sampInt:
        sampSine[time]=np.sin(twopi*time/sampInt)
```

```
          time=time+1
  # can be replaced by two calls to np.linspace
      sineWav=sampSine
      cnt=LFOrate-1
      while cnt>0:
          sineWav=np.append(sineWav,sampSine)
          cnt=cnt-1
      mod=sineWav*depth
  # tremolo
      return soundData*(1+mod)
```

9.2.6 Amplifier

The VCA controls the overall volume of the sound output.

Inputs	sound data, volume level
Output	amplified sound data

Type into ‘main3.py’

```
  # amplifier
  def VCA(soundData,volume=1):
      return soundData*volume
```

9.2.7 Displays and Output

Time (in number of samples) and frequency (in Hz(displays can be generated from the sound data from the VCO, VCF or VCA and this sound data then stored in a file.

Inputs	sound data, output file
Output	`None`

Type into ‘main3.py’

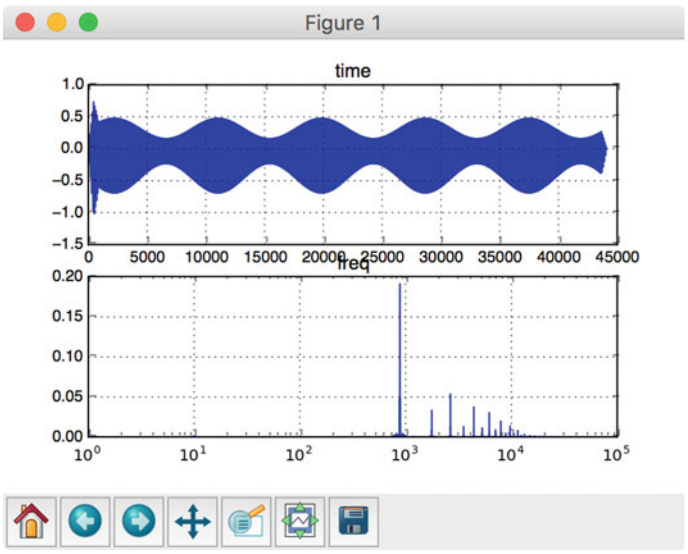
```
  # display and output
  def display(soundData,fileName='out'):
      plt.ion()
      plt.figure(1)
      plt.subplot(211)
      plt.plot(wavData[0:freqSamp])
      plt.title('time')
      plt.grid()
      plt.subplot(212)
      plt.semilogx()
      plt.title('freq')
```

```
    plt.grid()
    plt.plot(abs(fft.rfft(wavData))/32768)
    io.write(fileName+'.wav',freqSamp,soundData)
    return None
```

9.2.8 User Interface

The full synthesiser works with just calling the above functions in a nested fashion as in
`display(VCA(LFO(AEG(VCF(VCO())))))`
with or without default parameter values. `display` can be inserted before `VCO`, `VCF` or `VCA` depending on the output to be monitored. A full GUI requires the widgets described above plus the mixer GUI support function.

Using the default values and %matplotlib in the ipython console results in

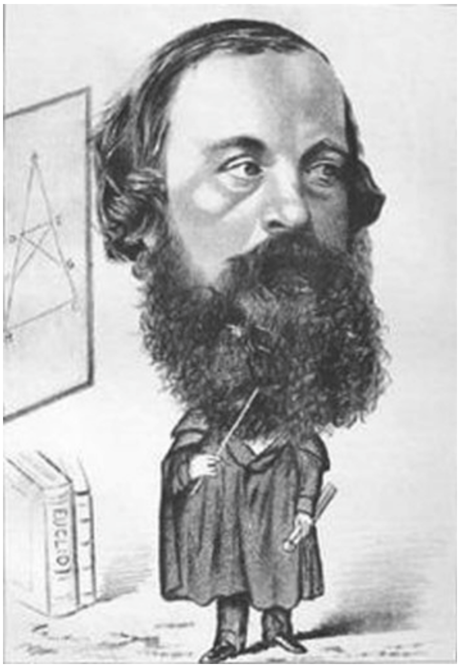


where enlarging the foot of each main spectral line shows the effect of the 5 Hz LFO modulation, and is probably also an artefact of performing a real FFT without a window function. The different height of the harmonics (spectral lines) reflect the mixture of sawtooth, square and triangle waveforms.

What you can then do is mix different combinations and multiples of `VCF`, `AEG` and `LFO` in any order, between the `VCA` and `VCO` outermost functions. This gives the sound producing capabilities of the Moog Modular synthesisers of the 1960s. All this from very simple synthesis functions!

Part IV
The Harmonograph—Victorian Pendulum Toy

Chapter 10
Hugh Blackburn (1823 to 1909)



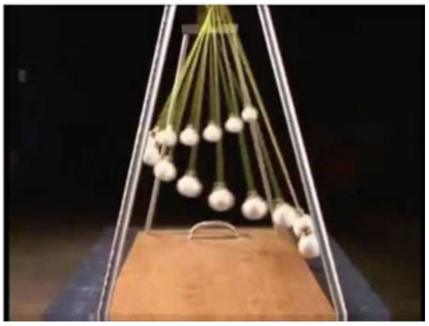
Hugh Blackburn, seen here in caricature, was Professor of Mathematics at Glasgow University for thirty years from 1849 to 1879. Whilst an undergraduate at Cambridge he designed a form of double pendulum, described in section below, which bears his name. This device was used to investigate harmonic intervals.

I. Gray, *Snake Charming—The Musical Python*,
DOI 10.1007/978-3-319-60660-6_10

10.1 Motion of a Damped Pendulum

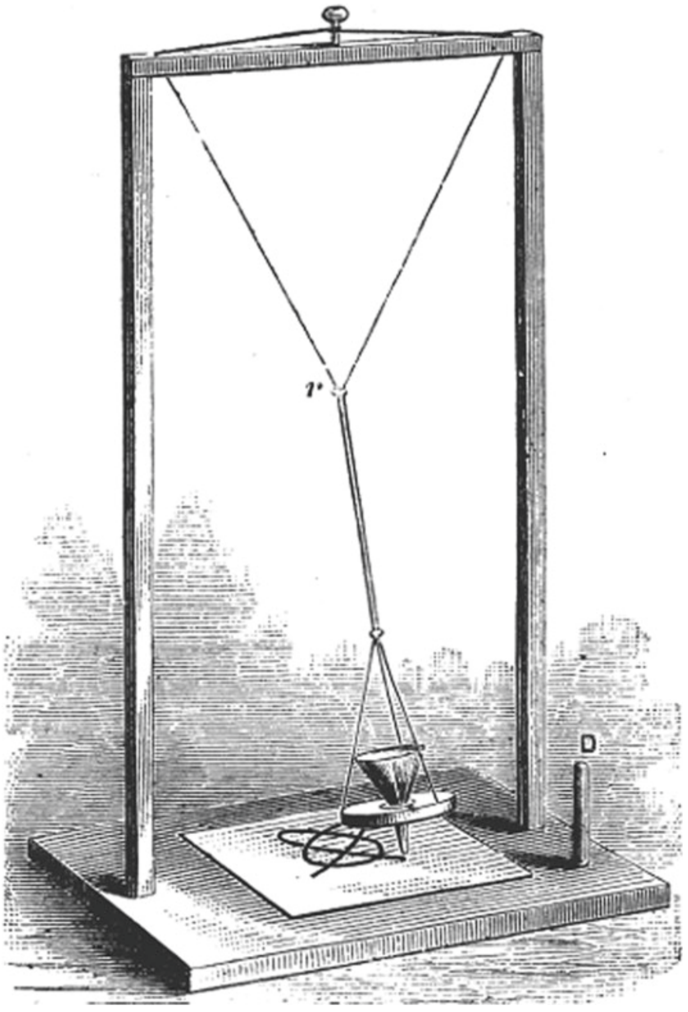
This section which is more mathematically challenging may be safely skipped, at least on first reading, without losing the theme of this chapter.

A simple pendulum, without friction, is a simple harmonic oscillator and so obeys, to a small angle approximation, the second order differential equation $\frac{d^2\theta}{dt^2} + \frac{g}{l} \cdot \theta = 0$, where θ is the angular displacement of the pendulum, l the pendulum's length and g is gravitational acceleration. This gives a period for the pendulum of $T_0 = 2 \cdot \pi \cdot \sqrt{\frac{l}{g}}$ by Huygen's law. This correspondence between length and period is illustrated by a device called the Pendulum Wave, in which fourteen pendulums of increasing length are displaced by the same amount then released, resulting in



This also illustrates the fundamental trigonometric nature of the solution to the differential equation $\theta(t) = \theta_0 \cdot \cos(\sqrt{\frac{g}{l}} \cdot t)$. Finally to account for damping caused by friction, at the pivot and air resistance, simply multiply this solution by a decreasing function of time such as $\exp(-d \cdot t)$, where d is a damping factor.

10.2 Blackburn's Double Pendulum



Blackburn's double pendulum, from Sound by John Tyndall, 1879.

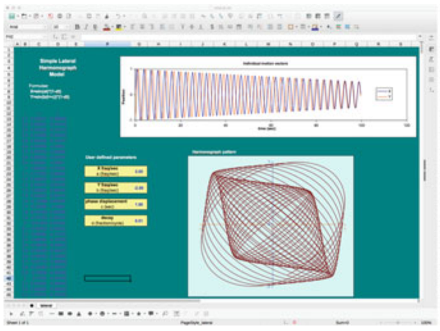
Blackburn's double pendulum is Y shaped. The top part, the arms of the Y, is generally made of a solid piece of wood, and pivoted so that it can only move in one axis. The bottom part, the stem of the Y, is attached to the upper part with some flexible material, like rope, allowing it the freedom to move in both axes. Attached to this lower part was a container of sand or ink to record the evolving patterns of the harmonic ratios.

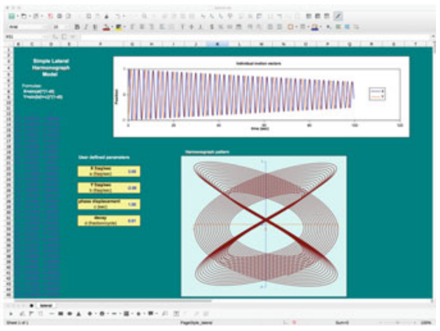
The major problem with this design is that the pendulum must be physically very large to record the finer pattern details caused by the decay of the pendulum motion. As an example see the twelve foot Blackburn pendulum built by Paul Wainwright in his barn, Please see book backmatter.

10.3 Harmonic Ratios—The Lateral Harmonograph

Clearly it was quite impractical to carry around a twelve foot Blackburn pendulum to Victorian drawing rooms, to entertain the other guests after a dinner party! The solution came in the form of a lateral harmonograph small enough to pack in a wooden carrying case. As well as legs to support the wooden case as a table, it had two metal pendulums with movable bobs. The pendulums were fitted in two gimbals at right angles to each other, thus ensuring orthogonal motion. One pendulum held a drawing surface and the other a recording pen. The bobs could be moved relative to each other to establish different harmonic ratios.

The illustrations below are respectively of a beat note, and a perfect fifth interval, and were made using Francis McConville's spreadsheet simulation of a lateral harmonograph, Please see book backmatter. This site also contains illustrations of lateral harmonographs.

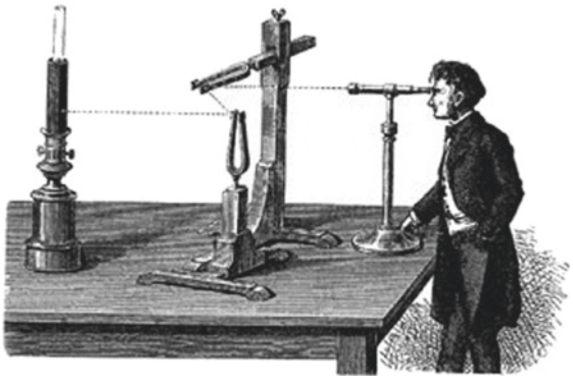




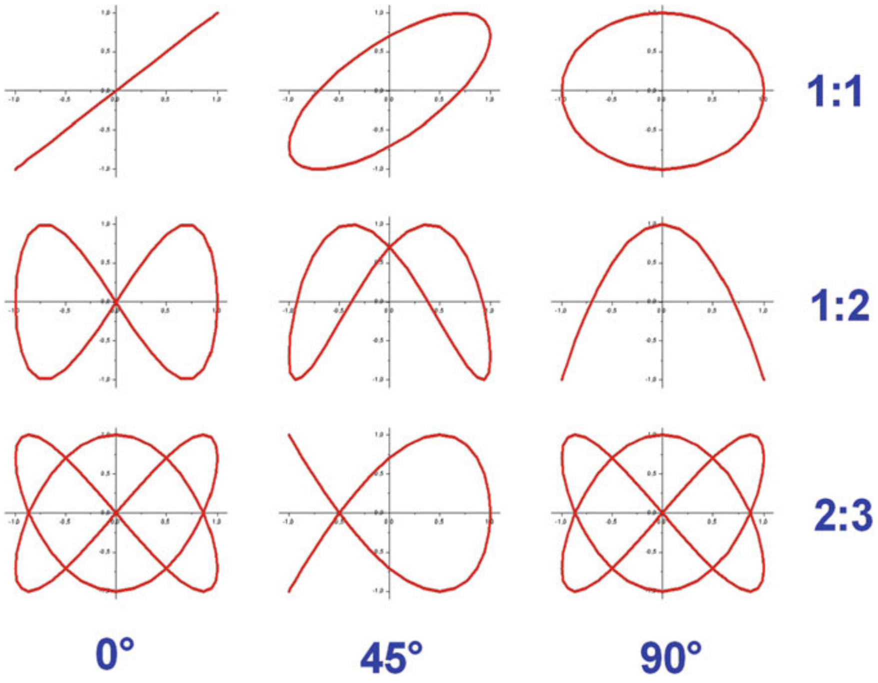
10.4 Parallels—Bowditch and Lissajous

It is almost a truism in science that the same idea is worked on independently by different people simultaneously, witness Newton and Leibniz on differential calculus. In the case of harmonic analysis Nathaniel Bowditch, the father of ocean navigation, in work later refined by Jules-Antoine Lissajous, worked on this study independently of Hugh Blackburn.

Nowadays, when the appearance of Lissajous' curves of harmonic ratios is familiar from oscilloscope screens it is very revealing to see his apparatus from nearly a century before such displays became commonplace. Two tuning forks were mounted at right angles to each other with each bearing a mirror on one of their arms. This arrangement then focussed light on to a telescope through which the curves could be observed.

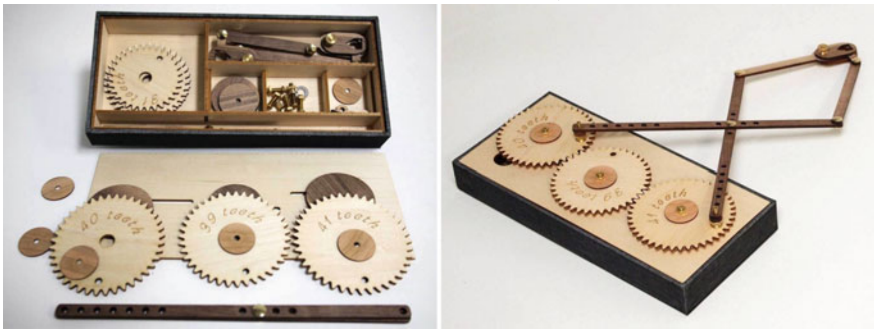


These curves are arranged by phase angle against harmonic ratio.



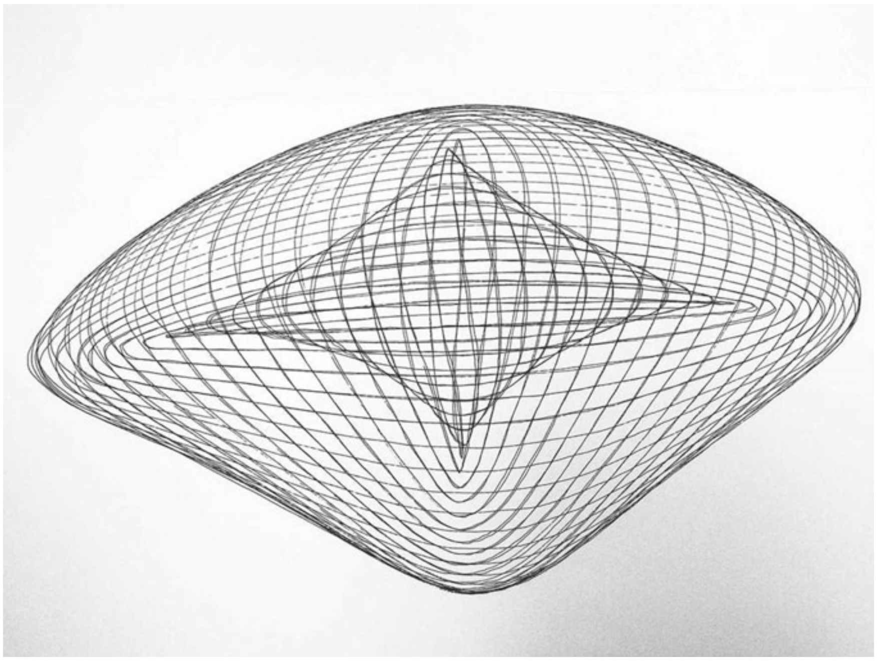
10.5 Of Gears and Motors—The Pintograph

Instead of varying pendulum length, by moving bobs, to set up the ratios for harmonic intervals, pintographs do this purely mechanically. This can either be through setting up gear ratios or varying stepper motor speeds. Resulting in their drawings being related to, but not identical to, the equivalent harmonograph drawings.
The pintograph illustrated below is the PrimoGraf from LEAFpdx, Please see book backmatter. It consists of seven co-prime gears which can be set up in pairs or triples to achieve different ratios. It is then cranked manually.



Joe Freedman, www.sarabande.com

The beat note illustrated below, with gear ratios 40:39:41, should be compared with that from the lateral harmonograph.



Joe Freedman, www.sarabande.com

Chapter 11
Project Code

This project will develop an animation of a lateral harmonograph where you can specify the harmonic ratios to be plotted. The various stages involved are

1. Plotting Lissajous' figures
2. Plotting two orthogonal damped pendulums
3. adding user interface control over harmonic ratios, expressed as rational numbers, to be plotted

11.1 New Language Features

Project Headers given in New Language Features are applicable only to this section.

11.1.1 Lissajous' Figures

Lissajous' curves are specified by the pair of parametric equations
$x(t) = A \cdot sin(a \cdot t + \phi)$ and $y(t) = B \cdot sin(b \cdot t)$ where

1. t is time
2. A and B are amplitudes taken as unity here
3. a and b are frequencies, a/b giving their harmonic ratio
4. ϕ is the phase offset between the two sinusoids

I. Gray, *Snake Charming—The Musical Python*,
DOI 10.1007/978-3-319-60660-6_11

11.1.1.1 Just Temperament Intervals

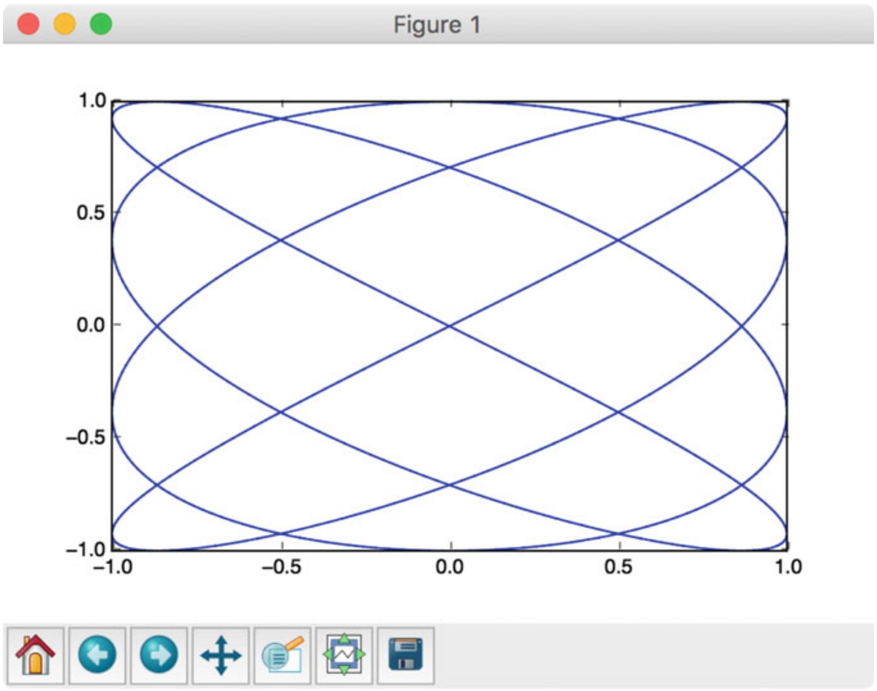
Just temperament is used here, instead of Equal temperament (where all semitones are $\sqrt[12]{2}$), as all the intervals can be expressed as rational numbers with integral numerators and denominators. These intervals are

1. Unison, ratio 1:1
2. Octave, ratio 2:1
3. Perfect Fifth, ratio 3:2
4. Perfect Fourth, ratio 4:3
5. Major Third, ratio 5:4
6. Minor Third, ratio 6:5
7. Major Tone, ratio 9:8
8. Minor Tone, ratio 10:9
9. Semitone, ratio 16:15

Enter the following test program into Python saving it as 'liss,py'.

```
import numpy as np
import matplotlib.pyplot as plt
def lissajous(t,a,b,phi):
    xt=np.sin(a*t+phi)
    yt=np.sin(b*t)
    return xt,yt
print(lissajous(1,4,3,np.pi/6))
size=int(np.pi*20000)
xs,ys=np.zeros(size),np.zeros(size)
a,b,phi=4,3,np.pi/3
for index in range(size):
    (xs[index],ys[index])=lissajous(index/10000,a,b,phi)
print(xs,ys)
plt.ion()
plt.plot(xs,ys)
```

The first highlight makes the program iterate over $2 \cdot \pi$ cycles, while the second highlighted `for` loop can of course be substituted by a `linspace` function. For a Perfect Fourth interval (4:3) the output should look like (after %maplotlib in IPython).

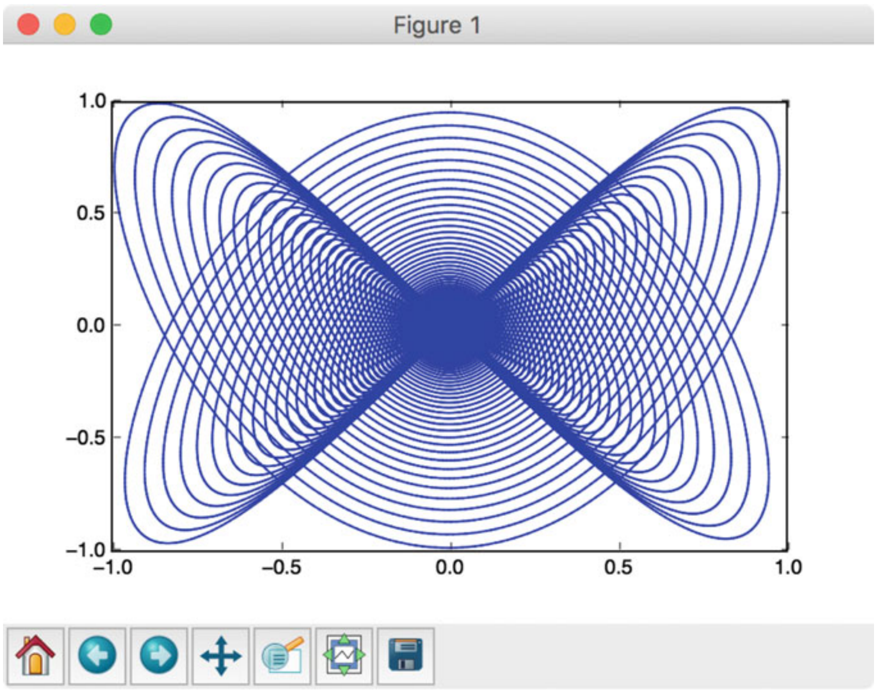


11.1.2 *Damped Orthogonal Pendulums*

Enter the following test program into Python saving it as ‘damp.py’

```
# harmonograph with damping factor 1/100
import numpy as np
import matplotlib.pyplot as plt
def damped(t,a,b,phi):
    xt=np.sin(a*t+phi)*np.exp(-t/100)
    yt=np.sin(b*t)*np.exp(-t/100)
    return xt,yt
size=int(np.pi*2000000)
xs,ys=np.zeros(size),np.zeros(size)
a,b,phi=2,3,np.pi
for index in range(size):
    (xs[index],ys[index])=damped(index/10000,a,b,phi)
print(xs,ys)
plt.ion()
plt.plot(xs,ys)
```

The first two highlights multiply by the damping factor,while the third highlight iterates over $200 \cdot \pi$ cycles. The diagram shows a harmonograph of a perfect fifth (after %maplotlib in IPython).



11.1.3 *Harmonic Ratios as Fractions*

Python expresses rational numbers in the module `fractions`. Enter the following test program into Python saving it as 'ratio.py'

```
# harmonic ratios as fractions
from fractions import Fraction
unison=Fraction(1,1)
octave=Fraction(2,1)
perfect5th=Fraction(3,2)
perfect4th=Fraction(4,3)
major3rd=Fraction(5,4)
minor3rd=Fraction(6,5)
majortone=Fraction(9,8)
minortone=Fraction(10,9)
semitone=Fraction(16,15)
print(unison,octave,perfect5th,perfect4th,major3rd,
        minor3rd,majortone,minortone,semitone)
print(perfect5th.numerator,perfect5th.denominator)
```

Where `Fraction` is the constructor function, `numerator` and `denominator` the accessor functions for Rational numbers.

11.1.4 User Interface

The widgets from tkinter that will form the GUI will be

Overall	a single **Frame** to contain remaining widgets
Overall	a **Button** to start the program when all user entries, including default values, are completed
Intervals	a **Combobox** will allow selection of harmonic interval
Phase	a **Combobox** will allow selection of phase offset between the two sinusoids
Plot type	a **Combobox** will allow damping on or off to be chosen

11.1.5 Project Header

You will need to use some features of NumPy and Matplotlib for this project so enter here

```
import numpy as np
import matplotlib.pyplot as plt
```

11.2 The Code

The project Python code is thus a composite of the above subsections, with suitable modifications to accommodate the GUI.

11.2.1 Project Header

Type into 'main4.py'

```
from math import *
from fractions import Fraction
import numpy as np
import matplotlib.pyplot as plt
from tkinter import *
from tkinter import ttk
```

11.2.2 Orthogonal Polynomials

Two different pairs of orthogonal pendulums are defined

1. Lissajous an idealised system, without friction, where the pendulums move at a harmonic ratio and phase shift with respect to each other
2. Harmonograph a realistic system, with frictional damping, where the pendulums move at a harmonic ratio and phase shift with respect to each other

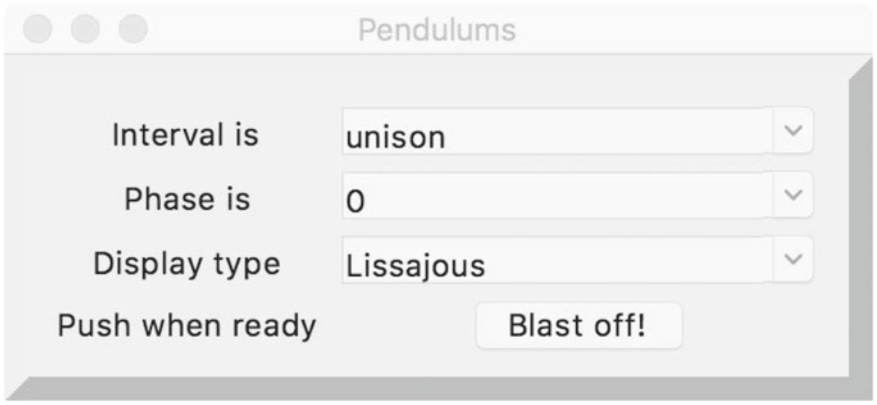
Inputs harmonic ratio as a fraction, phase shift in degrees, friction as Boolean
Output `None`

Type into ‘main4.py’

```
def orthogonal(ratio=Fraction(2,3),phase=90,friction=False):
    def undamped(a,b,phi):
        def lissajous(t):
            xt=np.sin(a*t)
            yt=np.sin(b*t+phi)
            return xt,yt
        size=int(np.pi*20000)
        xs,ys=np.zeros(size),np.zeros(size)
        for index in range(size):
            (xs[index],ys[index])=lissajous(index/10000)
        return (xs,ys)
    def damped(a,b,phi):
        def harmonograph(t):
            xt=np.sin(a*t)*np.exp(-t/100)
            yt=np.sin(b*t+phi)*np.exp(-t/100)
            return xt,yt
        size=int(np.pi*200000)
        xs,ys=np.zeros(size),np.zeros(size)
        for index in range(size):
            (xs[index],ys[index])=harmonograph(index/10000)
        return (xs,ys)
    a,b,phi=ratio.numerator,ratio.denominator,phase*np.pi/180
    (xs,ys)=damped(a,b,phi) if friction else undamped(a,b,phi)
    plt.ion()
    plt.figure(1)
    plt.plot(xs,ys)
    return None
```

11.2.3 *User Interface*

A simple GUI is developed to fill in the entries for the pendulums



for more sophisticated GUIs see Chapter 15.

for more sophisticated GUIs see Appendix A.
Type into ‘main4.py’

```
def getInterval(*args):
    intervalStr=interval.get()
    if intervalStr==‘unison’:
        intervalValue=Fraction(1,1)
    elif intervalStr==‘octave’:
        intervalValue=Fraction(1,2)
    elif intervalStr==‘perfect fifth’:
        intervalValue=Fraction(2,3)
    elif intervalStr==‘perfect fourth’:
        intervalValue=Fraction(3,4)
    elif intervalStr==‘major third’:
        intervalValue=Fraction(4,5)
    elif intervalStr==‘minor third’:
        intervalValue=Fraction(5,6)
    elif intervalStr==‘tone’:
        intervalValue=Fraction(8,9)
    else:
        intervalValue=Fraction(9,10)
    return intervalValue
def getPhase(*args):
    phaseValue=float(phase.get())
    return phaseValue
def getDisplay(*args):
    displayStr=display.get()
    displayValue=(displayStr==‘Harmonograph’)
    return displayValueh
def start(*args):
    i=getInterval()
    p=getPhase()
    f=getDisplay()
    orthogonal(i,p,f)
    return None
root=Tk()
root.title(‘Pendulums’)
frame=ttk.Frame(root,padding=10,borderwidth=10,relief=‘raised’)
frame.grid(column=0,row=0,sticky=(N,W,E,S))
ttk.Label(frame,text=‘Interval is’).grid(column=1,row=1)
ttk.Label(frame,text=‘Phase is’).grid(column=1,row=2)
ttk.Label(frame,text=‘Display type’).grid(column=1,row=3)
ttk.Label(frame,text=‘Push when ready’).grid(column=1,row=4)
intervalVar=StringVar()
intervalList=[‘unison’,‘octave’,‘perfect fifth’,
                ‘perfect fourth’, ‘major third’,
                ‘minor third’, ‘tone’, ‘semitone’]
interval[‘values’]=intervalList
interval.set(‘unison’)
interval.bind(‘<<ComboboxSelected>>’,getInterval)
interval.grid(column=2,row=1)
phaseVar=StringVar()
phaseList=[‘0’, ‘15’, ‘30’, ‘45’, ‘60’, ‘75’, ‘90’]
phase=ttk.Combobox(frame,textvariable=phaseVar)
phase[‘values’]=phaseList
phase.set(‘0’)
phase.bind(‘<<ComboboxSelected>>’,getPhase)
phase.grid(column=2,row=2)
displayVar=StringVar()
displayList=[‘Lissajous’, ‘Harmonograph’]
display=ttk.Combobox(frame,textvariable=displayVar)
display[‘values’]=displayList
display.set(‘Lissajous’)
display.bind(‘<<ComboboxSelected>>’, getDisplay)
display.grid(column=2,row=3)
btn=ttk.Button(frame,text=‘Blast off!’, command=start)
btn.grid(column=2,row=4)
root.mainloop()
```

Although this is a relatively simple GUI it illustrates the main features

1. different types of widget
2. grid geometry manager
3. bindings and control callback on events
4. the event loop

This GUI must be extended with a canvas widget to allow drawing, see Chap. 14.

Part V
Counterpoint à la Mode—Composing Music

Chapter 12
Johann Joseph Fux (1660 to 1741)

Fux was a prolific Austrian composer of the early Baroque period, however it is mainly for his music theoretical works on harmony and counterpoint that he is remembered today. This work covered tonal music from the 17th to late 19th centuries, the era of Common Practice. However, as will be shown, this can be adapted to cover modern jazz and rock genres in the Extended Common Practice.

I. Gray, *Snake Charming—The Musical Python*,
DOI 10.1007/978-3-319-60660-6_12

12.1 Gradus Ad Parnassum—Counterpoint

Fux's masterpiece Gradus ad Parnassum, steps to Parnassus, was originally published in 1725 and has remained in print ever since. Written in Latin, the Lingua Franca of its day, it consists of two parts

1. The Theory—covers intervals as a mathematical exercise, consonances, motions and rules for motion between intervals.
2. The Dialogue—between the master Aloysius (Palestrina) and his student Josephus (Fux) consisting of a dozen didactic problems per species. These problems cover prescribed cantus firmus above and below the accompanying melody line, in each of the permissible modes, and can be considered the Baroque equivalent of cryptic crossword puzzles.

The book was known to have been greatly admired by J.S.Bach, and that composers like Mozart and Beethoven, who worked out all the exercises, then influenced the whole of tonal music through its teachings.

12.1.1 Melody—Direct, Contrary and Oblique Motion

Motion can be considered the horizontal (temporal) aspect of the music, and of how two (or more) such melody lines move with respect to each other.

1. Direct motion—both melodic lines ascend or descend in pitch during the same time interval
2. Contrary motion—one melodic line ascends while the other descends in pitch during the same time interval
3. Oblique motion—one melodic line ascends or descends in pitch while the other melodic line remains at the same pitch during the same time interval

Fux presents four rules for allowable motions, however these can be simplified into a single rule—the only forbidden motion is direct into a perfect interval.

12.1.2 Harmony—Consonance and Dissonance

Harmony can be considered the vertical aspect of the music and what notes are viewed as being consonant or dissonant to each other. The consonance and dissonance interpretation will vary dependent on the genre or period under consideration, for instance Baroque or modern jazz.

1. the unison, major third, perfect fifth, major sixth and octave are considered consonant intervals
2. all other intervals, including the perfect fourth, are considered dissonant and only permitted as transitions between consonant intervals which resolve them

Transition rules and handling of dissonances will be covered in detail in Chap. 13.

12.1.3 Species Counterpoint

Fux's introduction of five species of counterpoint is the one still followed today. It comprises

1. Note against note—only checks for consonant intervals
2. Two notes against note—in addition checks motion between melodic lines
3. Four notes against note—in addition checks for dissonance transitions
4. Ligature (Syncopation)—adds time shifting between melodic lines
5. Florid—combines any of the first four species

Later composers would elaborate the third species so that it could consider melodies in triple, rather than just duple, time.

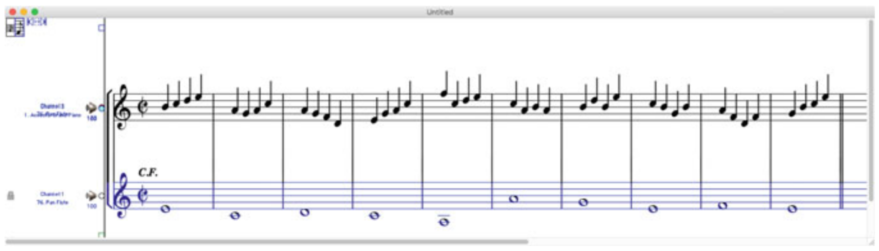
12.1.4 Modal Music

Although bearing the names of the earlier Greek (Ecclesiastical) modes, Fux uses them to refer to the starting and finishing tonic note in his cantus firmus. They are thus the current modal scales and comprise

1. Ionian—tonic C—corresponds to a Major scale
2. Dorian—tonic D
3. Phrygian—tonic E
4. Lydian—tonic F
5. Mixolydian—tonic G
6. Aeolian—tonic A—corresponds to a natural Minor scale

Note that Fux never uses the Locrian mode (tonic B) the darkest of all the modes because, transposed into the key of C, it has too many accidentals. The Locrian mode is however of great practical use in the composition of modern jazz counterpoint.

12.2 Strict Rules Allow Freedom of Composition



This exercise by Fux was composed and checked using Ars-Nova's Counterpointer software.

The example is of Fux's third species, cantus firmus below in the Phyrgian mode. Although there are still a few inadmissible leaps in the melody this example shows the fluidity achievable within strict counterpoint. Yes, you can compose dull, rote music with, or without, rules, let your imagination guide your choices. May the force be with you!

Chapter 13
Project Code

This project will allow the semi-automated composition of two part counterpoint in the style of Palestrina or the Baroque period. It has a number of sub-projects

1. develop a pink noise generator for the automatic creation of a cantus firmus in a user specified ecclesiastical mode
2. develop an efficient encoding of the rule set for user specified species of counterpoint
3. develop a list based representation of allowable voice leadings and a user specified choice of note
4. allow user to hear, save and playback this choice and the whole composition
5. allow user to modify consonant/dissonant groups and motion rules to reflect a particular genre such as modern jazz

13.1 The Colours of Noise

Noise can occur in many colours, including black, but only three are important musically

1. White describes noise with equal energy per frequency ($\propto \frac{1}{f^0}$) and is responsible for the high frequency background hiss of a detuned television or radio receiver. Its main use musically is to produce a cacophonous sound for mixing in with the synthesis of percussion.
2. Pink noise has equal energy per octave and so is $\propto \frac{1}{f^\alpha}, \alpha \approx 1$ and decays at $\frac{-3dB}{octave}$. It is self-similar (fractal) in nature having a blend of chaotic and correlated behaviour that makes it very useful in music. It can be applied both to time to generate rhythmic patterns, and to pitch to compose melodies which are tonal rather aleatoric (random) in nature.
3. Brown noise is that exhibited in the classic random (drunkard's) walk where every position reached is dependent on the immediately previous position. Its energy is $\propto \frac{1}{f^2}$ and so decays at $\frac{-6dB}{octave}$ thus making it generally too correlated for extensive use in music. 'Brown' refers to the Scottish botanist Robert Brown who observed

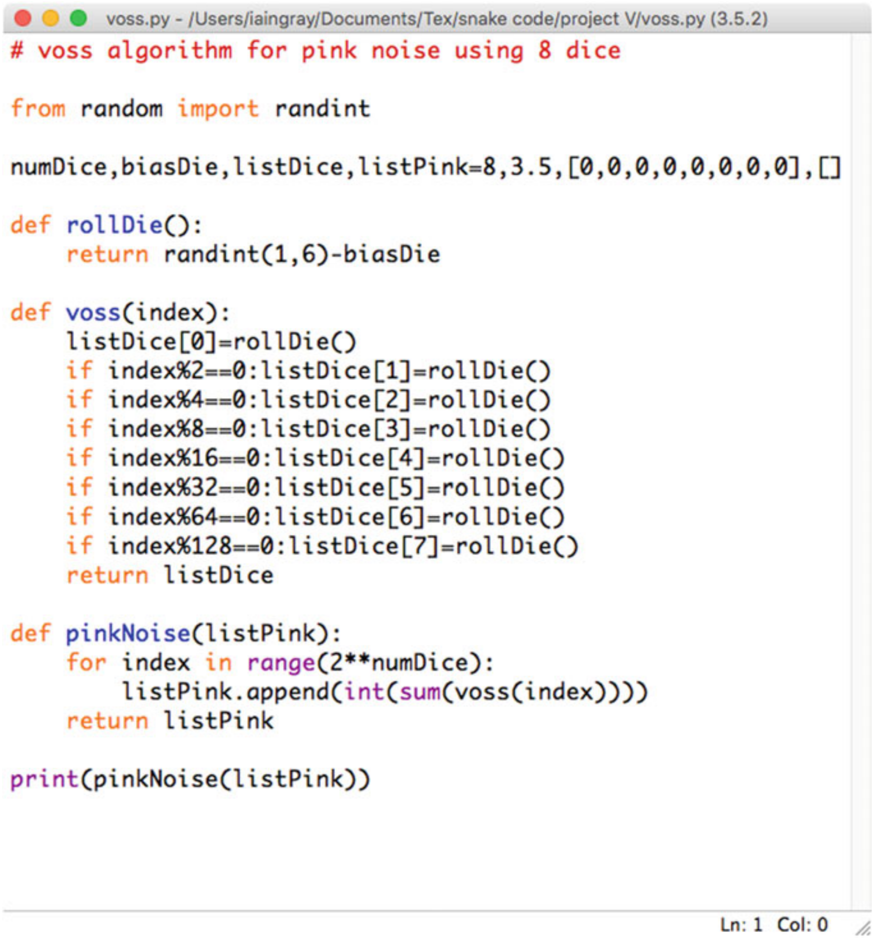
I. Gray, *Snake Charming—The Musical Python*,
DOI 10.1007/978-3-319-60660-6_13

random motion of pollen grains in water under a microscope. This Brownian motion was eventually explained by Albert Einstein as being due underlying molecular motion.

The functions available from importing Python's module `random` are adequate for noise generation. A very good 'dicey' algorithm for generating the memory properties of pink noise was invented by Richard Voss and described by Martin Gardner, see Bookbackmatter. In it you roll a number of dice and sum then to get your initial value of pink noise, then continue rolling and summing as follows

- roll die 'one' every throw
- roll die 'two' every second throw
- roll die 'three' every fourth throw
- roll die 'four' every eighth throw
- and so on with each subsequent die rolled half as often as the previous die

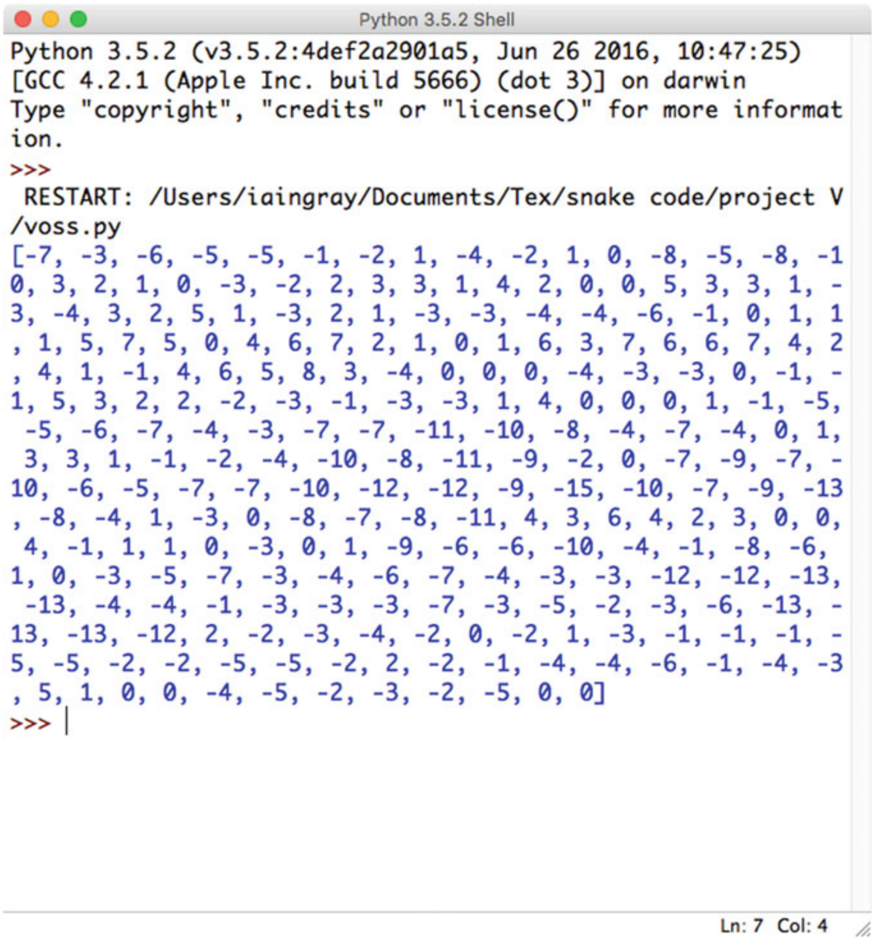
The small test program 'voss.py' and typical output illustrates this for eight dice,



voss.py - /Users/iaingray/Documents/Tex/snake code/project V/voss.py (3.5.2)

```
# voss algorithm for pink noise using 8 dice

from random import randint

numDice,biasDie,listDice,listPink=8,3.5,[0,0,0,0,0,0,0,0],[]

def rollDie():
    return randint(1,6)-biasDie

def voss(index):
    listDice[0]=rollDie()
    if index%2==0:listDice[1]=rollDie()
    if index%4==0:listDice[2]=rollDie()
    if index%8==0:listDice[3]=rollDie()
    if index%16==0:listDice[4]=rollDie()
    if index%32==0:listDice[5]=rollDie()
    if index%64==0:listDice[6]=rollDie()
    if index%128==0:listDice[7]=rollDie()
    return listDice

def pinkNoise(listPink):
    for index in range(2**numDice):
        listPink.append(int(sum(voss(index))))
    return listPink

print(pinkNoise(listPink))
```

Ln: 1 Col: 0



```
Python 3.5.2 (v3.5.2:4def2a2901a5, Jun 26 2016, 10:47:25)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "copyright", "credits" or "license()" for more informat
ion.
>>>
 RESTART: /Users/iaingray/Documents/Tex/snake code/project V
/voss.py
[-7, -3, -6, -5, -5, -1, -2, 1, -4, -2, 1, 0, -8, -5, -8, -1
0, 3, 2, 1, 0, -3, -2, 2, 3, 3, 1, 4, 2, 0, 0, 5, 3, 3, 1, -
3, -4, 3, 2, 5, 1, -3, 2, 1, -3, -3, -4, -4, -6, -1, 0, 1, 1
, 1, 5, 7, 5, 0, 4, 6, 7, 2, 1, 0, 1, 6, 3, 7, 6, 6, 7, 4, 2
, 4, 1, -1, 4, 6, 5, 8, 3, -4, 0, 0, 0, -4, -3, -3, 0, -1, -
1, 5, 3, 2, 2, -2, -3, -1, -3, -3, 1, 4, 0, 0, 0, 1, -1, -5,
 -5, -6, -7, -4, -3, -7, -7, -11, -10, -8, -4, -7, -4, 0, 1,
 3, 3, 1, -1, -2, -4, -10, -8, -11, -9, -2, 0, -7, -9, -7, -
10, -6, -5, -7, -7, -10, -12, -12, -9, -15, -10, -7, -9, -13
, -8, -4, 1, -3, 0, -8, -7, -8, -11, 4, 3, 6, 4, 2, 3, 0, 0,
 4, -1, 1, 1, 0, -3, 0, 1, -9, -6, -6, -10, -4, -1, -8, -6,
1, 0, -3, -5, -7, -3, -4, -6, -7, -4, -3, -3, -12, -12, -13,
 -13, -4, -4, -1, -3, -3, -3, -7, -3, -5, -2, -3, -6, -13, -
13, -13, -12, 2, -2, -3, -4, -2, 0, -2, 1, -3, -1, -1, -1, -
5, -5, -2, -2, -5, -5, -2, 2, -2, -1, -4, -4, -6, -1, -4, -3
, 5, 1, 0, 0, -4, -5, -2, -3, -2, -5, 0, 0]
>>> |
```

Where the numbers represent semitone distance from the tonic over a two octave range.

13.2 New Language Features

Project Headers given in New Language Features are applicable only to this section. Lists, trees as a list of lists, and list manipulation functions will be used extensively throughout this project.

13.2.1 Project Header

You will need to use some features of NumPy and Matplotlib for this project so enter here

```
import numpy as np
import matplotlib.pyplot as plt
```

13.2.2 Sound Storage

Arrays from NumPy will be used to hold the numerical samples of the sound. These will be turned into WAV files for input, output and offline processing of these sounds. Add the following line to the project header `from scipy import io`

13.2.3 MIDI—Musical Instrument Digital Interface

MIDI is a wide ranging technical standard allowing instruments such as synthesisers to be connected together within a common framework. Notes are specified by integers in the range 0–127 with successive numbers representing semitone intervals. Middle C (C4) is MIDI note number 60, with C3 equalling 48 and C5 equalling 72.

The small Python program in ‘ET.py’ enables the calculation of Equal Temperament note frequencies from a knowledge of the reference A frequency and the number of semitones difference to the note, e.g. concert A is 440 Hz and MIDI note 69, middle C (C4) is MIDI note 60 and hence -9 semitones.

```
root=2**(1/12)
def ET(refA,semitones):
    freq=refA
    if semitones<0:
        for st in range(abs(semitones)):
            freq=freq/root
    elif semitones>0:
        for st in range(semitones):
            freq=freq*root
    return freq
```

13.2.4 *Composing the Cantus Firmus with Pink Noise Generation*

A minor modification to 'voss.py' will ensure that the pink noise output is tonal, rather than chromatic, by specifying only the allowable semitone spacings over a two octave range. Enter the following Python code as 'cantus.py' with correct indentations

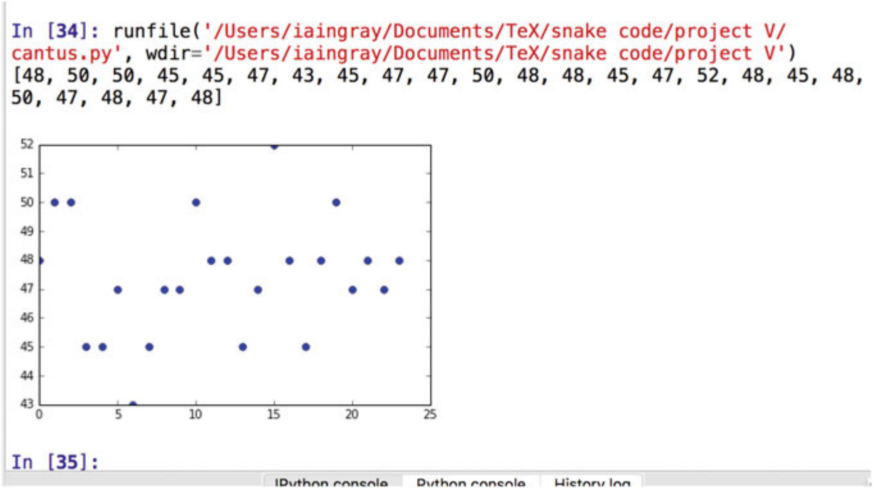
```
import numpy as np
import matplotlib.pyplot as plt
midiC3=48
# voss algorithm for pink noise using 8 dice
from random import randint
numDice,biasDie,listDice,listPink=8,3.5, [0,0,0,0,0,0,0,0],[midiC3]
tonalInt,indexRange=[-12,-10,-8,-7,-5.-3,-1,0,2,4,5,7,9,11,12],24
def rollDie():
    return randint(1,6)-biasDie
def voss(index):
# establishing which dice to roll
    listDice[0]=rollDie()
    if index%2==0:listDice[1]=rollDie()
    if index%4==0:listDice[2]=rollDie()
    if index%8==0:listDice[3]=rollDie()
    if index%16==0:listDice[4]=rollDie()
    if index%32==0:listDice[5]=rollDie()
    if index%64==0:listDice[6]=rollDie()
    if index%128==0:listDice[7]=rollDie()
    return listDice
def pinkNoise(listPink):
    pink=0
# mapping over required number of notes
    for index in range(indexRange):
        pink=int(sum(voss(index)))
# filtering out non-tonal intervals
        while pink not in tonalInt:
            pink=int(sum(voss(index)))
        listPink.append(pink+midiC3)
    listPink.append(midiC3)
    return listPink
print(pinkNoise(listPink))
plt.plot(listPink,'bo')
plt.show()
```

The three new features are list related

1. list declarations as a dynamic structure indexed from 0, with `[midiC3]` being the tonic for starting and ending the piece
2. list elements can be summed together
3. lists can be formed by appending new elements on to the end

The only difference from 'voss.py' is the additional 'while' loop to filter out non-tonal intervals. Along with the 'for' loop this constitutes the map-filter paradigm in functional programming.

Illustrated below is a typical IPython session output from the above program where the graph's vertical axis is the MIDI note number.



```
In [34]: runfile('/Users/iaingray/Documents/TeX/snake code/project V/
cantus.py', wdir='/Users/iaingray/Documents/TeX/snake code/project V')
[48, 50, 50, 45, 45, 47, 43, 45, 47, 47, 50, 48, 48, 45, 47, 52, 48, 45, 48,
50, 47, 48, 47, 48]
```

```
In [35]:
```

13.2.5 *Sound Generation*

There are two extensions to the earlier presentation of square and triangle wave sound presented in Chap. 9.

1. Stereo Output—this will allow the simultaneous output of the Cantus Firmus and the counterpoint on two separate channels.
2. Continuous Frequency—this will allow notes on a tonic scale to be output at their correct frequencies.

13.2.5.1 Stereo Output

Enter the following test program into Python saving it as 'stereo.py'.

```
# cantus firmus 2 second
import numpy as np
from numpy import linspace, append
from scipy.io.wavfile import write
freqSamp=44100
duration=2
stereoNote=np.empty([2,duration*freqSamp],float)
freqNote=441
numRpts=freqSamp//freqNote
print(freqNote,numRpts)
halfRpts=numRpts//2
firstHalf=linspace(1.0,1.0,halfRpts,False).astype(float)
```

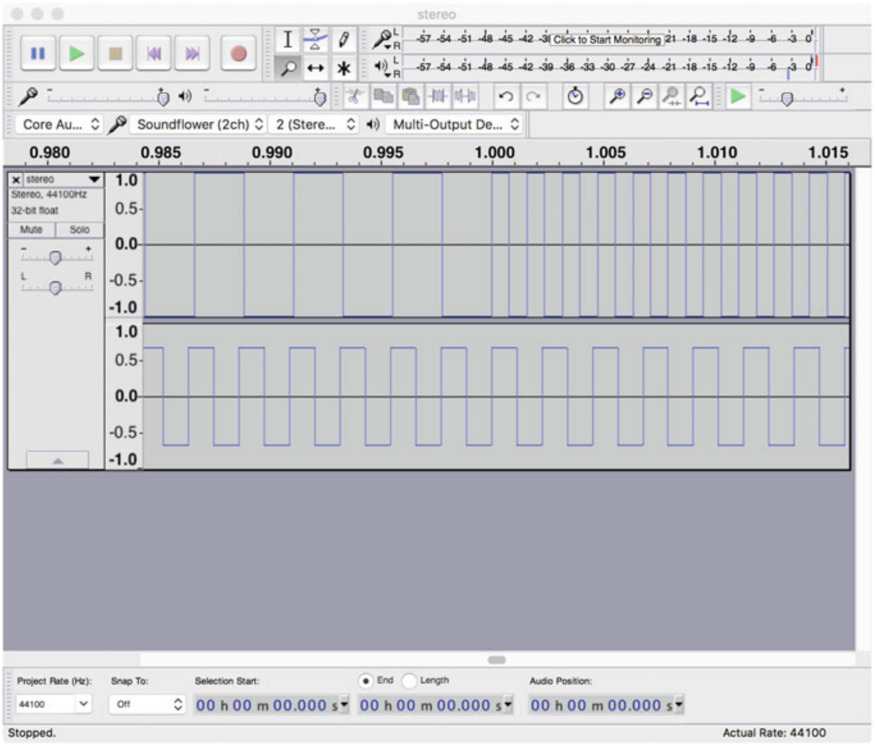
```
lastHalf=linspace(-1.0,-1.0,halfRpts,False).astype(float)
squData=append(firstHalf,lastHalf)
rpts=duration*freqNote-1
squNote=squData
while rpts>0:
    squNote=append(squNote,squData)
    rpts=rpts-1
print(squNote.shape)
# fourth up 1 second
duration=1
freqNote=630
numRpts=freqSamp//freqNote
print(freqNote,numRpts)
halfRpts=numRpts//2
firstHalf=linspace(1.0,1.0,halfRpts,False).astype(float)
lastHalf=linspace(-1.0,-1.0,halfRpts,False).astype(float)
fourthData=append(firstHalf,lastHalf)
rpts=duration*freqNote-1
fourthNote=fourthData
while rpts>0:
    fourthNote=append(fourthNote,fourthData)
    rpts=rpts-1
print(fourthNote.shape)
# octave down 1 second
duration=1
freqNote=225
numRpts=freqSamp//freqNote
print(freqNote,numRpts)
halfRpts=numRpts//2
firstHalf=linspace(1.0,1.0,halfRpts,False).astype(float)
lastHalf=linspace(-1.0,-1.0,halfRpts,False).astype(float)
octaveData=append(firstHalf,lastHalf)
rpts=duration*freqNote-1
octaveNote=octaveData
while rpts>0:
    octaveNote=append(octaveNote,octaveData)
    rpts=rpts-1
print (octaveNote.shape)
# stereo 2 seconds
oct4th=append(octaveNote,fourthNote)
level=0.67
squNote=level*squNote
print(oct4th.shape)
stereoNote=np.array((oct4th,squNote))
stereoNote=stereoNote.transpose()
print('stereo',stereoNote.shape,stereoNote.ndim)
print(stereoNote)
write('stereo.wav',freqSamp,stereoNote)
```

The code is in four sections each headed by a highlighted comment.

The first three sections construct three different frequencies and durations of square wave as before. Allocation of memory for storing all three is provided by the `empty` method applied to the `stereoNote` array.

The final section sets the Cantus Firmus output level to 2/3, and uses `transpose` to put `stereoNote` in the correct shape for stereo output.

The display shows the stereo output with the counterpoint on the upper channel, and the Cantus Firmus on the lower channel.



13.2.5.2 Continuous Frequency

This section contains two parts

1. common code applicable both to synthesis and counterpoint for setting up the scales and multipliers
2. keeping the sampling rate constant but varying the internal waveform data for output in stereo for counterpoint

Enter the following test program into Python saving it as ‘scale.py’.

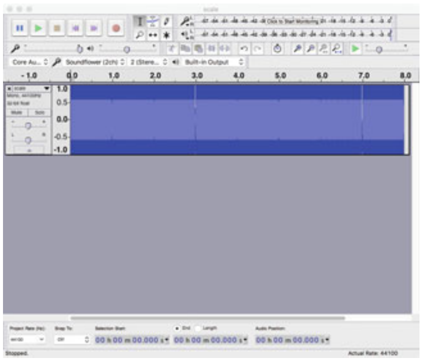
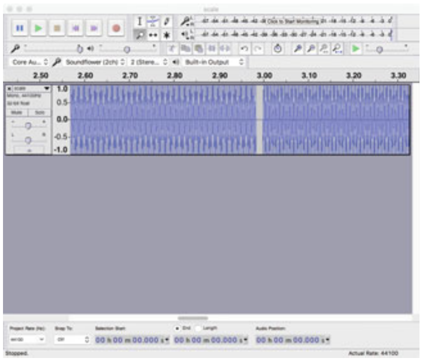
```
# scale playing
from math import floor
from numpy import linspace, append, zeros
from scipy.io.wavfile import write
```

```
freqSamp,rptsList,freqList,zeroList=44100,[],[],[]
ionianET=[261.63,293.66,329.63,349.23,392.00,440.00,493.88,523.25]
for note in ionianET:
    flNote=floor(freqSamp/note)
    flNote=flNote if flNote%2==0 else flNote-1
    rptsList.append(flNote)
    freqList.append(floor(note))
    zeroList.append(freqSamp-floor(note)*flNote)
print(rptsList,freqList,zeroList)
scale=[]
# triangle oscillator and padding
for count in range(8):
    halfRpts=rptsList[count]//2
    firstHalf=linspace(-1.0,1.0,halfRpts,False).astype(float)
    lastHalf=linspace(1.0,-1.0,halfRpts,False).astype(float)
    triData=append(firstHalf,lastHalf)
    rpts=freqList[count]-1
    triNote=triData
    while rpts>0:
        triNote=append(triNote,triData)
        rpts=rpts-1
    scale=append(scale,triNote)
    padding=zeros(zeroList[count]).astype(float)
    scale=append(scale,padding)
write('scale.wav',freqSamp,scale)
```

The code is in two sections each headed by a highlighted comment.

1. The Ionian scale in C4 in equal temperament is used with these frequencies. The floating point frequencies are truncated to the nearest even integer to allow for triangular waveforms. Zero padding is used to ensure that each note is of the same duration..
2. This generates triangular waves as previously in Chap. 9 and then appends zero padding.

The illustrations show the seamless joins in a scale over eight seconds, and an enlargement of the zero padding at three seconds.

13.2.6 Sound Output of Cantus Firmus

The following Python program in ‘socf.py’ uses the list output from ‘cantus.py’, the continuous frequencies of ‘scale.py’ and a dictionary of MIDI notes and their frequencies from ‘ET.py’ to allow playback of a generated Cantus Firmus.

```
# Cantus Firmus playing
from math import floor
from numpy import linspace, append, zeros
from scipy.io.wavfile import write
freqSamp,rptsList,freqList,zeroList=44100,[],[],[]
cantusFirmus=[48,50,50,45,45,47,43,45,47,47,50,48,
                48,45,47,52,48,45,48,50,47,48,47,48]
rangeET=36: 65.41, 38: 73.42, 40: 82.41, 41: 87.31,
        43: 98, 45: 110, 47: 123.47, 48: 138.81,
        50: 146.83, 52: 164.81, 53: 174.61,
        55: 196, 57: 220, 59: 246.94, 60: 261.63
notes=[]
for idx in cantusFirmus:
    note=rangeET[idx]
    notes.append(note)
for note in notes:
    flNote=floor(freqSamp/note)
    flNote=flNote if flNote%2==0 else flNote-1
    rptsList.append(flNote)
    freqList.append(floor(note))
    zeroList.append(freqSamp-floor(note)*flNote)
print(rptsList,freqList,zeroList)
CF=[]
# triangle oscillator and padding
for count in range(24):
    halfRpts=rptsList[count]//2
    firstHalf=linspace(-1.0,1.0,halfRpts,False).astype(float)
    lastHalf=linspace(1.0,-1.0,halfRpts,False).astype(float)
    triData=append(firstHalf,lastHalf)
    rpts=freqList[count]-1
    triNote=triData
    while rpts>0:
        triNote=append(triNote,triData)
        rpts=rpts-1
    CF=append(CF,triNote)
    padding=zeros(zeroList[count]).astype(float)
    CF=append(CF,padding)
write('CF.wav',freqSamp,CF)
```

A dictionary is an association list where `key:data` pairings are separated by commas and enclosed in {}s. The first highlight shows the creation of a MIDI note : frequency dictionary. While the second highlight shows the extraction of frequency for a given MIDI note.

13.2.7 The Rules in Python

The rules considered are those for consonance and counterpoint motion only at this stage.

In Python MIDI notes are used modulo 12 for checking all rule compliances for the proposed counterpoint.

13.2.7.1 Allowable Consonances

The notes for the Cantus Firmus (CF) are subtracted (modulo 12) from those of the Counterpoint (CP) and checked for membership of `consonance=[0,4,7,9]`. In Python this is

```
if (CP-CF)%12 in consonance:
```

13.2.7.2 Allowable Motions

A list of the Cantus Firmus dynamics (rising, falling or staying same) is compared with that of two successive counterpoint notes to determine the motion type of Direct, Contrary or Oblique. The only motion forbidden is Direct into a Perfect consonance (Unison, Perfect Fifth or Octave). In Python this is `(counter-cantus)$12 not in [0,7]`.

13.2.7.3 Allowable Dissonance Transitions

In third species only the third of four notes is allowed to be dissonant `not in [0,4,7,9]` provided it is `in [2,5,11]` and follows in step. This will be added in Chap. 14.

13.2.8 Output of Whole, Half and Quarter Notes for Counterpoint

The Cantus Firmus is played in whole notes (semi-breves), whereas the Counterpoint uses fractions of these depending on its species.

1. First Species (note against note) continues to use whole notes (semi-breves)
2. Second Species (two note against one) uses half notes (minims)
3. Third Species (four notes against one) uses quarter notes (crotchets)

These different note lengths will be generated as in Continuous Frequency above with the appropriate scales for durations and paddings. The `freqSamp` remains at 44100 Hz.

13.2.9 User Interface and Sound

The widgets from tkinter that will form the GUI will be

Overall	a single **Frame** to contain remaining widgets
Overall	a **Button** to start the program when all user entries, including default values, are completed
Cantus Firmus	two **Button**s for generating and playing the Cantus Firmus
Species	a **Radiobutton** to select from Species 1–3 of counterpoint
Counterpoint	**Button**s to record and erase, move up and down allowable notes or remain on the same note, and playback current notes

13.2.9.1 GUI Helper Up Down Buttons

In order to have the Up and Down buttons moving by degrees in the modal scale use is made of dictionaries. Type the following into 'updown.py'

```
up,down=1,-1
upIonian={0:2,1:1,2:2,3:1,4:1,5:2,6:1,7:2,8:1,9:2,10:1,11:1}
downIonian={0:1,1:1,2:2,3:1,4:2,5:1,6:1,7:2,8:1,9:2,10:1,11:2}
def upDown(dirn,note):
    note12=note%12
    if dirn==up:
        return note+upIonian[note12]
    return note-downIonian[note12]
print(upDown(up,64))
print(upDown(down,67))
```

Note that `return` is unconditional so no `else` is required.

13.3 The Code

The project Python code is thus a composite of the above subsections, with suitable modifications to accommodate the GUI. The lists passed as parameters will either be as MIDI notes or frequencies for the Cantus Firmus (CF) or Counterpoint (Cpt). Sound output will be in stereo using square wave oscillators.

13.3.1 Project Header

Type into 'main5.py'

```
from math import *
from fractions import Fraction
import numpy as np
from tkinter import *
from tkinter import ttk
freqSamp=44100
allowFreqs=[22050,11025,7350,4410,3675,3150,2450,2205,
            1575,1470,1225,1050,882,735,630,525,490,
            450,441,350,315,294,245,225,210,180,175,
            150,147,126,105,98,90,75,70,63,60,50,45,42,35,30]
allowMods=[30,25,21,18,15,14,10,9,7,6,5,3,2,1]
midiC2,midiC3,midiC4,midiC5,midiC6=36,48,60,72,84
midiET={36:65.41,38:73.42,40:82.41,41:87.31,
        43:98.00,45:110.00,47:123.47,
        48:130.81,50:146.83,52:164.81,53:174.61,
        53:196.00,57:220.00,59:246.94,
        60:261.63,62:293.66,64:329.63,65:349.23,
        67:392.00,69:440.00,71:493.88,
        72:523.25,74:587.33,76:659.26,77:698.46,
        79:783.99,81:880.00,83:987.77,
        84:1046.50}
tonalInt=[-12,-10,-8,-7,-5,-3,-1,0,2,4,5,7,9,11,12]
consonance,perfect,imperfect=[0,4,7,9],[0,7],[4,9]
speciesDur={1:44100,2:22050,3:11025}
up,down=1,-1
upIonian={0:2,1:1,2:2,3:1,4:1,5:2,6:1,7:2,8:1,9:2,10:1,11:1}
downIonian={0:1,1:1,2:2,3:1,4:2,5:1,6:1,7:2,8:1,9:2,10:1,11:2}
```

13.3.2 Generating the Cantus Firmus with Pink Noise

This composes the Cantus Firmus bars in semi-breves used throughout as a basis for the Counterpoint.

Inputs	central tonic note, number of bars
Output	complete midiCF

Type into ‘main5.py’

```
def cantusFirmus(centreC,numBars):
    def rollDie():
        return randint(1,6)-biasDie
    def voss(index):
        listDice[0]=rollDie()
        if index%2==0:listDice[1]=rollDie()
        if index%4==0:listDice[2]=rollDie()
        if index%8==0:listDice[3]=rollDie()
        if index%16==0:listDice[4]=rollDie()
        if index%32==0:listDice[5]=rollDie()
        if index%64==0:listDice[6]=rollDie()
        if index%128==0:listDice[7]=rollDie()
        return listDice
    def pinkNoise(listPink):
        pink=0
        for index in range(indexRange):
            pink=int(sum(voss(index)))
# filtering out non-tonal intervals
            while pink not in tonalInt:
                pink=int(sum(voss(index)))
            listPink.append(pink+centreC)
        listPink.append(centreC)
        return listPink
    numDice,biasDie,listDice,listPink=8,3.5,[0,0,0,0,0,0,0,0],[centreC]
    indexRange=numBars-2
    listPink=pinkNoise(listPink)
    midiCF
    return midiCF
```

13.3.3 Cantus Firmus Dynamics

Generate a list from CF showing its underlying dynamics for determination of Cpt motions.

Input	midiCF
Output	list of underlying dynamics

Type into ‘main5.py’

```
def dynamicCF(midiCF):
    dynCF=[]
    for index in range(len(midiCF)-1):
        dir=midiCF[index+1]-midiCF[index]
        if dir>0:
            dyn='rise'
        elif dir<0:
            dyn='fall'
        else:
            dyn='same'
        dynCF=np.append(dynCF,dyn)
    return dynCF
```

13.3.4 Rules of Counterpoint

Consonance, Motion and Dissonance are handled by a knowledge of the Cantus Firmus, current note and dynamics, proposed Counterpoint note and the Species. This returns an accept Boolean value.

Inputs	indexCF, midiCF (list), dynCF (list), midiCpt (partial list), noteCpt (candidate)
Output	accept (Boolean), list of MIDI Counterpoint notes

Type into 'main5.py'

```
    def rules(indexCF,midiCF,dynCF,midiCpt,noteCpt):
        def motion():
            prevCpt=midiCpt[-1]
            motCpt=noteCpt-prevCpt
            currDynCF=dynCF[indexCF]
            if currDynCF=='rise' and motCpt>0:
                return 'direct'
            elif currDynCF=='fall' and motCpt<0:
                return 'direct'
            elif currDynCF=='same' and motCpt==0:
                return 'direct'
            else:
                return 'indirect'
        currInt=(noteCpt-midiCF[indexCF])%12
        accept=False
        if motion()=='indirect':
            if currInt in consonant:
                midiCpt=np.append(midiCpt,noteCpt)
                accept=True
        elif currInt in imperfect:
            midiCpt=np.append(midiCpt,noteCpt)
            accept=True
        return (accept,midiCpt)
```

13.3.5 Converting MIDI to Frequency

Before being output as sound the MIDI notes must be converted to frequencies.

Input	list of MIDI note
Output	list of frequencies

Type into ‘main5/py’

```
def midi2freq(midiList):
    freqList=[]
    for note in midiList:
        freqList.append(midiET[note])
    return freqList
```

13.3.6 Handling Whole, Half and Quarter Notes

Cantus Firmus and Species 1 Counterpoint are in whole (semi-breve) notes, Species 2 Counterpoint is in half (minim) notes and Species 3 Counterpoint in quarter (crotchet) notes. These durations are handled by a dictionary `speciesDur` giving the various durations of the sample rate `freqSamp`.

Inputs	list of frequencies, species
Output	sound data1 list

Type into ‘main5.py’

```
def freqData(freqList,species):
    def padSaw(freqNote):
        numRpts=modSamp//floor(freqNote)
        sawData=np.linspace(1.0,-1.0,numRpts,False).astype(float)
        rpts=floor(freqNote)-1
        sawNote=sawData
        while rpts>0:
            sawNote=np.append(sawNote,sawData)
            rpts=rpts-1
        padding=np.zeros(modSamp-len(sawNote))
        sawNote=np.append((sawNote,padding))
        return sawNote
    modSamp=speciesDur[species]
    noteData=[]
    for note in freqList:
        noteData=np.append(notedata,padSaw(note))
    return noteData
```

13.3.7 Stereo Output

Final output involves combining the wavCF and wavCpt into a two dimensional array, list of lists, transposing this and outputting this as a stereo '.wav' file.

Inputs	wavCF, wavCpt, output file name
Output	`None`

Type into 'main5.py'

```
def stereo(wavCF,wavCpt,stereoFile):
    stereoNotes=np.array((wavCF,wavCpt))
    stereoNotes=stereoNote.transpose()
    io.write(stereoFile+'.wav',freqSamp,stereoNotes)
    return None
```

13.3.8 User Interface

Control the counterpoint up/same/down key GUI **Button** presses to only allow modal degree steps.

Inputs	direction key, original MIDI note
Output	resultant MIDI note

Type into 'main5.py

```
def upDown(dirn,note):
    note12=note%12
    if dirn==up:
        return note+upIonian[note12]
    return note-downIonian[note12]
```

Edit the counterpoint list.

Inputs	counterpoint MIDI list, MIDI note
Output	counterpoint MIDI list

Type into 'main5.py'

```
def addNote(cptList,note):
    cptList.append(note)
    return cptList
def delNote(cptList):
    del cptList[-1]
    return cptList
```

Display the Cantus Firmus and Counterpoint.

Inputs	cantus firmus and counterpoint MIDI lists, species of counterpoint
Output	None

Type into ‘main5 .py’

```
  def display(CFList,cptList,species):
      plt.plot(CFList,’bo’)
      xlst,lcpt=[],len(cptList)
      if species==1:
          plt.plot(cptList,’r.’)
      elif species==2:
          idx=0
          while idx<lcpt/2:
              xlst.append(idx)
              idx=idx+0.5
          plt.plot(xlst,cptList,’r.’)
      else:
          idx=0
          while idx<lcpt/4:
              xlst.append(idx)
              idx=idx+0.25
          plt.plot(xlst,cptList,’r.’)
      plt.show()
```

The full functionality of a Counterpoint assistant is contained in ‘main5.py’. This can be further enhanced with a full GUI as in Chap. 14.

Part VI
On Safari

Chapter 14
Where Next?

This chapter is an ordered list of extensions and enhancements to each project left as exercises to the reader.

14.1 Generic Header

The following in 'header.py' will prove useful in any extension

```
# generic header delete those not needed
import time
from math import *
from fractions import *
from random import randint
import numpy as np
import scipy.special as spe
import scipy.io.wavfile as io
import scipy.signal as sig
import scipy.fftpack as fft
import matplotlib.pyplot as plt
import matplotlib.animation as anim
from mpl_toolkits.mplot3d import Axes3D
from tkinter import *
from tkinter import ttk
import tkinter.colorchooser as col
freqSamp=44100
allowFreqs=[22050,11025,7350,4410,3675,3150,2450,2205,
            1575,1470,1225,1050,882,735,630,525,490,
            450,441,350,315,294,245,225,210,180,175,
            150,147,126,105,98,90,75,70,63,60,50,45,42,35,30]
allowMods=[30,25,21,18,15,14,10,9,7,6,5,3,2,1]
midiC2,midiC3,midiC4,midiC5,midiC6=36,48,60,72,84
midiET={36:65.41,38:73.42,40:82.41,41:87.31,
        43:98.00,45:110.00,47:123.47,
        48:130.81,50:146.83,52:164.81,53:174.61,
```

I. Gray, *Snake Charming—The Musical Python*,
DOI 10.1007/978-3-319-60660-6_14

```
          53:196.00,57:220.00,59:246.94,
          60:261.63,62:293.66,64:329.63,65:349.23,
          67:392.00,69:440.00,71:493.88,
          72:523.25,74:587.33,76:659.26,77:698.46,
          79:783.99,81:880.00,83:987.77,
          84:1046.50}
  tonalInt=[-12,-10,-8,-7,-5,-3,-1,0,2,4,5,7,9,11,12]
  consonance,perfect,imperfect=[0,4,7,9],[0,7],[4,9]
  speciesDur={1:44100,2:22050,3:11025}
  up,down=1,-1
  upIonian={0:2,1:1,2:2,3:1,4:1,5:2,6:1,7:2,8:1,9:2,10:1,11:1}
  downIonian={0:1,1:1,2:2,3:1,4:2,5:1,6:1,7:2,8:1,9:2,10:1,11:2}
```

14.2 Part II—Visualising Sound

Develop a fully animated circular drumhead that allows you to specify the point to hit the drum and then to hear how it sounds. The various stages involved are

1. animating two dimensional wave propagation using Bessel functions
2. extending this this to three dimensional wave propagation over a circular membrane noting that the radial components are as specified in 1 above
3. adding user interface control over fundamental frequency and vibration mode of drum.

Basic animation can be enhanced by

GUI	adding a GUI using techniques from Chap. 9.
Sound	when an enhanced percussive synthesiser is available add drum sounds.
Animation	consider the functions included in `matplotlib.animation` to see if they can offer smoother animation.

14.3 Part III—Creating Sound

The basic synthesiser can be expanded to a full synthesiser by the following steps in order for each module

GUI	complete GUI functionality as in Chap. 13.
VCO	implement full frequency using the methods of Chap. 13.
VCO	add pulse width modulation to the square wave oscillator by allowing its mark:space ratio (`firsthalf lastHalf`) to vary from 1:1.
VCO	allow variable note length and tempo using the methods of Chap. 13 to allow percussion synthesis.

VCO	add noise generators for white and pink noise using the methods of Chap. 13.
VCO	add an all-pass filter after each oscillator to allow phase changing. This should be done by rotating the entire '.wav' data as rotating an individual waveform will introduce clicks between waves not matching up.
VCF	add full frequency input for the cutoff frequency.
VCF	implement high-pass, band-pass and band-stop filters from `scipy.signal`.
VCF	add resonance to the filter. This is an involved problem in Digital Signal Processing but you could replace the `scipy.signal.butter` Butterworth filter with a `scipy.signal.cheby1` Chebyshev type I filter specifying suitable ripple in the passband. As an alternative to `scipy.signal` many good filter design tools. such as FilterDesignLab-IIR shown at foot of this chapter.
EG	add a second envelope generator to control the cutoff frequency of the VCF.
EG	normally the ADSR is controlled by a key pressed on and off, simulate this with a button widget in the GUI.
EG	add extra envelope stages beyond the four of ADSR.
LFO	allow targeting of the VCO to create Frequency Modulation (FM) as vibrato, using the techniques for continuous frequencies given in Chap. 13.
LFO	use waveforms other than a sine wave for modulation.
Displays	add a `scipy.signal.spectogram` to monitor the harmonic content over time of the waveform.
GUI	add GUI widgets to accommodate new functionality.
Duophony	add two note polyphony using the techniques for stereo sound generation given in Chap. 13.
Miscellaneous	add a Sequencer for recording and playback of melodies, using the techniques for scale generation given in Chap. 13.

Visit https://wiki.python.org/moin/PythonInMusic to see what is available in for instance MIDI interfacing for an external keyboard.

14.4 Part IV—Visualising Harmony

Visualisation can be enhanced by

GUI	add canvas widget for drawing as in Matplotlib examples.
GUI	add a beating interval to range of intervals.
Sound	add playback of the interval using the techniques for stereo sound generation given in Chap. 13.

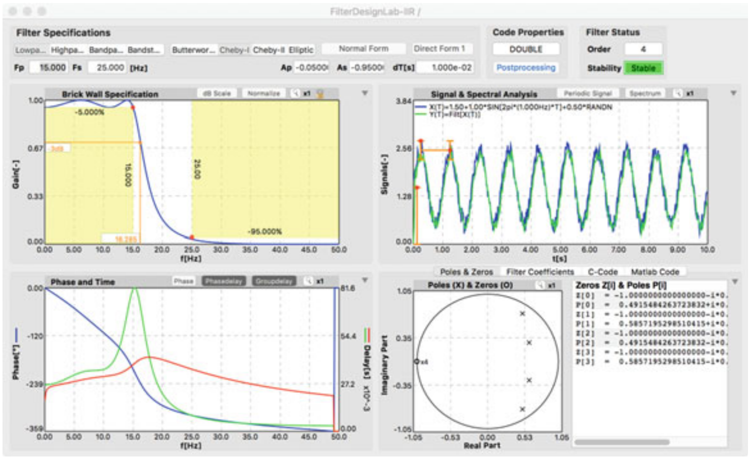
Animation	add a 'slowed down' animation of the displayed graphs using the techniques given in Chap. 5.
GUI	add GUI widgets to accommodate new functionality.

14.5 Part V—Composition

Basic Counterpoint can be enhanced as follows

GUI	complete GUI functionality as in Chap. 13.
Cantus Firmus	add Cantus Firmus above as well as below by swapping the note octave ranges.
Modes	add a complete set of modes beyond the Ionian with their tonics as in Chap. 12.
Rules	add dissonance handling and disallow leaps of more than a perfect fifth to the consonance and motion rules already included.
Species	add species 4 (ligature) and 5 (florid). Syncopation is achieved by rotating the Cantus Firmus list of notes.
Consonance	extend allowable consonances to reflect those of jazz scales.
Motion	extend allowable motions to reflect those of jazz scales.
GUI	add GUI widgets to accommodate new functionality.

FilterDesignLab-IIR



Fp 15.000 Fs 25.000 [Hz] Ap -0.0500 As -0.9500 dT[s] 1.000e-02

Order 4 Stability Stable

X(T)=1.50+1.00*SIN(2pi*(1.000Hz)*T)+0.50*RANDN
Y(T)=Filt(X(T))

Zeros Z[i] & Poles P[i]

```
Z[0]  = -1.0000000000000000-i*0.
P[0]  =  0.4915484263723832+i*0.
Z[1]  = -1.0000000000000000-i*0.
P[1]  =  0.5857195298510415+i*0.
Z[2]  = -1.0000000000000000-i*0.
P[2]  =  0.4915484263723832-i*0.
Z[3]  = -1.0000000000000000-i*0.
P[3]  =  0.5857195298510415-i*0.
```

Matthias Kretschmann - author of FilterDesignLab-IIR corresponding wedpage (http://www.mtk-digital.com/fdl/)

Curriculum Vitae

Iain Gray AFIMA Hons. BSc (Mathematics and Computational Science) Dundee

My background is that I have recently retired, after 35 years, as a Principal Systems Engineer working as a mathematician in radar research for a leading avionics company (Ferranti which latterly became Selex ES). I have over forty years programming experience and have worked in all the major paradigms. These comprise functional/declarative languages (Common Lisp and Prolog), procedural languages (Fortran, C and Pascal), "combination" languages (Scheme and Python) and object oriented languages (Simula and Ada).
1979–81 Navigation Systems Division—compiler development and implementation
1981–88 Display Systems Division—aircraft moving map displays and digital mapping
1988–90 Joint Venture—development of an aircraft mission management aid
1990–2014 Radar Systems Division—design and analysis of tracking systems
Papers presented
1989 NATO AGARD Toulouse
"Towards a Mission Management Aid" (jointly with J Catford)
1992 NATO AGARD Edinburgh
"Planning for Air to Air Combat"
1997 Radar 97 Edinburgh
"Effects of Earth's Curvature on Radar Tracking"
2002 GRS Radar Conference Bonn
"Advanced Tracking Systems" (jointly with Dr. B Spratt)
2004 Kalman Filtering Conference Reading
"Behavioural Analysis of Kalman Filters" (jointly with Dr. B Spratt)

I. Gray, *Snake Charming—The Musical Python*,
DOI 10.1007/978-3-319-60660-6

<u>Institute of Mathematics and its Applications (IMA)</u>
1982–84 Scottish Branch Council
mid 1980s Elected Associate Fellow (AFIMA)
1986–88 Main Council

Appendix
Internet Links

These websites have been useful in preparing the document and checking the results.

Part I

1. Python homepage https://www.python.org
2. Python downloads https://www.python.org/downloads/
3. Python documentation https://docs.python.org/
4. Tcl/Tk downloads http://www.activestate.com/activetcl/downloads
5. Anaconda https://www.continuum.io/downloads
6. PyQt https://riverbankcomputing.com/software/pyqt/intro
7. PySide https://wiki.qt.io/PySide
8. wxPython http://www.wxpython.org
9. Python(x,y) http://python-xy.github.io
10. Audacity http://www.audacityteam.org

Part II

1. Mark Kac https://en.wikipedia.org/wiki/Mark_Kac
2. Mark Kac's original paper http://www.maa.org/sites/default/files/pdf/upload_library/22/Chauvenet/Kac68chv.pdf
3. Analysis of paper https://en.wikipedia.org/wiki/Hearing_the_shape_of_a_drum
4. Drumhead modes https://en.wikipedia.org/wiki/Vibrations_of_a_circular_membrane
5. Vibration modes http://www.acs.psu.edu/drussell/Demos/MembraneCircle/Circle.html
6. Matplotlib tutorial https://www.labri.fr/perso/nrougier/teaching/matplotlib/
7. scipy drum example https://docs.scipy.org/doc/scipy/reference/tutorial/special.html
8. Matplotlib graphics examples http://matplotlib.org/examples/index.html

I. Gray, *Snake Charming—The Musical Python*,
DOI 10.1007/978-3-319-60660-6

Part III

1. Fourier https://en.wikipedia.org/wiki/Joseph_Fourier
2. Fourier transform https://en.wikipedia.org/wiki/Fourier_transform
3. Bob Moog https://en.wikipedia.org/wiki/Robert_Moog
4. Subtractive synthesis https://documentation.apple.com/en/logicstudio/instruments/index.htmlchapter=A
5. Moog website http://www.moogmusic.com
6. Arturia website https://www.arturia.com
7. SignalScope http://www.faberacoustical.com/apps/mac/signalscope/
8. SignalSuite http://www.faberacoustical.com/apps/mac/signalsuite/
9. Filter design http://www.mtk-digital.com/fdl/http://www.mtk-digital.com/fdl/
10. Prime factorisation http://factornumber.com/?page=44100
11. Python music packages https://wiki.python.org/moin/PythonInMusic
12. tkinter tutorial http://www.tkdocs.com

Part IV

1. Blackburn https://en.wikipedia.org/wiki/Hugh_Blackburn
2. Harmonograph https://en.wikipedia.org/wiki/Harmonograph
3. Blackburn pendulum http://paulwainwrightphotography.com/pendulum_gallery_video_page.shtml courtesy of http://paulwainwrightphotography.com
4. Drawing harmonic ratios http://www.fxmtech.com/harmonog.html courtesy of Francis McConville
5. Lissajous' curves https://en.wikipedia.org/wiki/Lissajous_curve
6. Just tuning https://en.wikipedia.org/wiki/Just_intonation
7. PrimoGraf http://leafpdx.bigcartel.com/product/primograf-drawing-machine0

Part V

1. Fux https://en.wikipedia.org/wiki/Johann_Joseph_Fux
2. Counterpoint https://en.wikipedia.org/wiki/Counterpoint
3. Counterpointer http://www.ars-nova.com/cp/
4. Coloured noise https://en.wikipedia.org/wiki/Colors_of_noise
5. Tuning http://www.rollingball.com/images/HT.pdf
6. MIDI https://en.wikipedia.org/wiki/MIDI

Bibliography

These books have proven useful in preparing the document and checking the results.

Part I

- M Lutz, "Python Pocket Reference", O'Reilly, 2014
- J Chan, "Python: Learn Python in One Day and Learn It Well", Learn Coding Fast, 2014
- The above is only representative of many good Python tutorial books

- JM Stewart, “Python for Scientists”, Cambridge University Press, 20144
- A good reference on object orientated Python.

Part III

- EO Brigham, “The Fast Fourier Transform and Its Applications” Prentice Hall, 1988
- M Puckette, “The Theory and Technique of Electronic Music”, WSPC, 2007
- A Farnell, “Designing Sound”, MIT Press, 2015
- A Kamenov, “Digital Signal Processing for Audio Applications”, CreateSpace Independent Publishing Platform, 2014
- R Allred, “Digital Filters for Everyone”, Creative Arts & Sciences House, 2015
- T Pinch and F Trocco, “Analog Days: The Invention and Impact of the Moog Synthesizer”, Harvard University Press, 2004
- BK Shepard, “Refining Sound: A Practical Guide to Synthesis and Synthesizers”, Oxford University Press, 2013.

Part IV

- A Ashton, “Harmonograph: A Visual Guide to the Mathematics of Music”, Wooden Books, 2005
- LP Pook, “Understanding Pendulums”, Springer, 2011
- J Tyndall, “Sound: A Course of Eight Lectures Delivered at the Royal Institution of Great Britain”, Adamant Media Corporation, 2005.

Part V

- JJ Fux, “The Study of Counterpoint: From Johann Joseph Fux’s Gradus Ad Parnassum”, Norton, 1965
- D Tymoczko, “A Geometry of Music: Harmony and Counterpoint in the Extended Common Practice”, Oxford University Press, 2011
- J Collins, “Counterpoint and How to Use It in Your Music”, Echo Pier Publishing, 2012
- RF Voss and J Clarke, “1/f noise” in music: Music from “1/f noise”, Journal of Acoustical Society of America, 63, 1918, pp 258–263
- M Gardner, “Fractal Music, Hypercards and More.... Mathematical Recreations from ‘Scientific American’ magazine”, WH Freeman, 1991
- M Hewitt, “Musical Scales of the World”, The Note Tree, 2013.