Creating interactive, dynamic, visual illustrations for teaching with high-level software tools

Cand. scient. thesis

Morten Wang Fagerland

Department of informatics

University of Oslo
2003

Preface

This paper is the thesis for the Cand.scient. (MSc) degree at the Department of informatics, University of Oslo, for Morten Wang Fagerland. The teaching supervisor for this work has been Professor Hans Petter Langtangen.

Oslo, October 2003

Contents

Part I

Introduction

Chapter 1

Abstract

Physical processes and algorithms that are dynamic in nature can be visualised by creating illustrations of moving objects. When used in teaching situations, such illustrations are an alternative to drawing rough figures on a blackboard or using an overhead. With a high degree of interactive control through a graphical user interface, the impact of parameters (physical or mathematical) on a system can be investigated with immediate results. This will often lead to a quicker way to learn and to increased understanding.

The purpose of this paper is to present a set of software tools (modules) that can be used to create illustrations with abstract, high-level programming using Python and Tkinter/Pmw/Blt. In addition several applications from subjects such as mechanics, informatics and numerical analysis are considered. The applications will show that using Python and these tools will make it possible for non-experts to create interactive, dynamic, visual illustrations.

Chapter 2

Get up and running

2.1 Online documentation

It is strongly recommended that you download this file

`http://folk.uio.no/mortenfa/thesis.tar.gz`

unzip and unpack it. Follow the instructions in this chapter and view the documentation found in the "doc/" folder. The documentation (*html* files) and this paper contain the same information. All the software presented here should then be available.

2.2 Necessary software

This software should be downloaded and installed first:

- Python and Tkinter (all in one at `www.python.org`). The most recent version (Python 2.3) is recommended, but most of the code here should work with earlier (and probably later) versions as well.
- Python megawidgets (`http://pmw.sourceforge.net`)
- The BLT Toolkit (`http://sourceforge.net/projects/blt`). Warning: the BLT toolkit seems to be somewhat poorly maintained. There may be problems combining different versions of BLT, PMW and Python, and the installation is troublesome. I have therefore used BLT in only two applications: Runge-Kutta and Newton's method.

Further discussion on BLT/Pmw may be found in the Pmw-general mailing list:

`http://lists.sourceforge.net/lists/listinfo/pmw-general`

2.3 Installation

The python interpretor needs to find the files in the modules folder. The parent folder (named "thesis") should therefore be placed in a folder that is included in the $PYTHONPATH variable. So, either add the current path to the variable or move the folder to an existing $PYTHONPATH path.

Installation of the Pmw widgets:

- Locate the Pmw installation folder (named "Pmw/Pmw_1_1/" in Pmw 1.1). From now on this will be referred to as "pmw/".
- Copy the "MultiListBox/PmwMultiListBox.py" file to the "pmw/lib/" folder.
- Open the file called "pmw/lib/Pmw.def" and include "MultiListBox" in the "_widgets" tuple defined at the top of the file.
- Copy the "MultiListBox/MultiListBox.py" file to the "pmw/demos/" folder. This will automatically add the MultiListBox demo to the "All.py" demo.
- Copy the "MultiListBox/MultiListBox_text.py" file to the "pmw/docsrc/text/" folder.
- Copy the "MultiListBox/MultiListBox.gif" file to the "pmw/docsrc/images/" folder.
- Copy the "MultiListBox/MultiListBox_test.py" file to the "pmw/tests/" folder. This will automatically add the MultiListBox test to the "All.py" test.
- Execute the "pmw/docsrc/createmanuals.py" file to add the MultiListBox reference page to the local Pmw manual.

Substitute ProgressBarDialog for MultiListBox and repeat the steps above.

You should hardcode the path to the root Pmw folder when using the MultiListBox and ProgressBarDialog widgets. Include the following lines of code (before importing Pmw) in your applications:

```
# Import Pmw from this directory tree.
import sys
sys.path[:0] = ['../../..']
```

Adjust the actual path when necessary. The example above implies that the Pmw root folder is positioned three levels below the application file. If this file is called "app.py" we may have a tree structure like this:

```
.../Python2.2/Pmw/
.../Python2.2/A/B/C/app.py
```

2.4 How to run the programs

Test the installation by trying one of the files in the demos folder. The "two_springs.py" file is the only one not using any files from the modules folder so you should try at least one of the other files as well.

To run the fluidflow and matrix applications, go to the approriately named folder and execute the "main.py" file. The other applications have executable files with the same name (for example: bucket-sort → "bucketsort.py").

Go to the "pmw/demos/" folder and execute the "All.py" file. If the installation of the Pmw widgets was successful you should see "MultiListBox" and "ProgressBarDialog" in the listbox.

Chapter 3

Why use Python as the main tool?

Python is a high-level object-oriented scripting language with a very clean and readable syntax. It is equipped with powerful built in data types, making it easy to built nested data structures, which is a greatly valued ability. If execution speed is essential, program code may be written in Fortran, C or C++, and made available through normal Python code. With toolkits such as Tkinter and Pmw, professional looking graphical user interfaces (GUIs) are easily created. This latter ability is exactly what we need to create visual illustrations. Python is also platform independent, and may be ported between different operating systems (see section 18.3 for some notes on this).

There exists a lot of software systems that offer tools to create animations and algorithm visualisations. This thesis is primarily interested in creating illustrations that visualise how algorithms (or dynamic processes) work, while most other systems seem to be more interested in visualising the results of algorithms. Python's open source policy and a growing number of active developers suggest that Python is going to increase its popularity in the science community.

A lot more can be said about the advantages of Python and of scripting in general (see for example [13] and [14]). Tutorials and references on Python, Tkinter and Pmw are plentiful, but these are lifesavers: [2], [3], [4], [5] and [6].

Chapter 4

How the reference pages (html) were created

4.1 Overview

This paper is an adaptation of the reference pages which accompany the software package. The reference pages were originally written in *html* to make them suitable for online browsing. This chapter explains the tools used to created the *html* code.

Each reference page is created by means of an ordinary text file (called the source file) and the `createdocs.py` file. The source file may contain tags that are either standard html tags or one of the special tags defined below. The `createdocs.py` file processes the source file and creates the resulting html file. The name of the html file will be the same as the source file, but with the suffix ".html" instead of ".txt".

4.2 Tags

None of the tags are case sensitive and must be written (including possible arguments like "<section>section name") on separate lines in the source file. The exceptions are the <func> and </func> tags that may be written anywhere. A paragraph tag <P> is inserted if an empty line is encountered (but only one at a time).

To separate the tags upper case letters have been used for standard *html* tags and lower case letters for the new tags.

- <`file`>`pyfile` - Use this if there is a python file connected to the reference page.
- <`title`>`header` - Create a H2 sized page header with a blue underline. If the <`file`> tag was used, set the title to "Reference:" + `pyfile` and create the H4 sized **File** header containing a link to the file. If the <`file`> tag was not used, set the title to "Reference:" + the name of the source file without the ".txt" suffix.
- <`modules`> - Scan the `pyfile` for lines including the text "`import \s+`" and list them under the **Modules** header.
- <`contents`> - Create a list of contents. This will contain the section names as defined by the <`section`> tags. A link to each section will be created.

- <section>name - Create a new section with a H4 sized header given by name. Insert a horizontal ruler above the header.
- </section> - End the current section.
- </section code> - End the current section. Insert the function or class from pyfile with the same name as the section.
- </section file> - End the current section. Insert the whole pyfile.
- <code> - Start a code paragraph. Treat the following text as python code. Use the <PRE> tag to display the text and make sure the "<" and ">" symbols are drawn correctly. If this tag does not occupy a separate line in the source file, the tag is treated as the standard html tag.
- </code> - End a code paragraph. If this tag does not occupy a separate line in the source file, the tag is treated as the standard html tag.
- <delimeter> - Insert the delimeter image centered on the page.
- <func> - Start displaying a function on the form **funcname(***arg1, arg2, ...***)**. The funcname + "(" is written with bold font, the arguments in italics, and the ")" in bold. The above statement is coded: "<func>funcname(arg1, arg2, ...)</func>".
- </func> - End displaying a function.

The foot of the page will automatically inserted at the end of the html file.

4.3 Code

```
#!/bin/sh
""":"
exec python $0 ${1+"$@"}
"""

# This will enable the program to be run from idle
import os
import sys
dirname,basename = os.path.split(sys.argv[0])
os.chdir(os.path.normpath(dirname))

import re

# The source files
files = (
         'index.txt',
         'abstract.txt',
         'installation.txt',
         'python.txt',
         'reference_howto.txt',
         'modules_coords.txt',
         'modules_dialogs.txt',
         'modules_functions.txt',
         'modules_misc.txt',
```

```
          'modules_programflow.txt',
          'modules_widgets.txt',
          'applications_fluidflow.txt',
          'applications_matrix.txt',
          'applications_rungekutta.txt',
          'applications_sorting_introduction.txt',
          'applications_sorting_baseclasses.txt',
          'applications_sorting_template.txt',
          'applications_sorting_bucket.txt',
          'applications_sorting_insertion.txt',
          'applications_sorting_merge.txt',
          'applications_sorting_quick.txt',
          'applications_sorting_shell.txt',
          'applications_vibrations_introduction.txt',
          'applications_vibrations_basegui.txt',
          'applications_vibrations_system.txt',
          'applications_vibrations_free.txt',
          'applications_vibrations_forced.txt',
          'applications_equationsolvers_introduction.txt',
          'applications_equationsolvers_baseclasses.txt',
          'applications_equationsolvers_bisection.txt',
          'applications_equationsolvers_newton.txt',
          'applications_equationsolvers_secant.txt',
          'pmw_introduction.txt',
          'application_template.txt',
          'demos_baseprefs.txt',
          'demos_function.txt',
          'demos_programflow.txt',
          'demos_smoothrectangle.txt',
          'demos_spring.txt',
          'demos_two_springs.txt',
          'bibliography.txt',
)

class CreateHTML:
    def __init__(self, sourcefile):
        self.sourcefile = sourcefile
        self.noNewParagraph = 1
        self.codeOn = 0
        self.funcOn = 0
        self.currentSectionName = ''

        self.read_source_file()
        self.create_html_file()
        self.parse_source()
        self.close_html_file()

    def read_source_file(self):
        file = open(self.sourcefile, 'r')
        self.sourcelines = file.readlines()
        file.close()

    def create_html_file(self):
        self.file = open('../doc/' + self.sourcefile[:-4] + '.html', 'w')
        self.file.write('<HTML>\n\n')

    def parse_source(self):
        tags = (('<file>', 'self.get_filename'),
                ('<title>', 'self.title'),
```

```
            ('<modules>', 'self.modules_list'),
            ('<contents>', 'self.contents_list'),
            ('<section>', 'self.start_section'),
            ('</section', 'self.end_section'),
            ('<code>', 'self.start_code_paragraph'),
            ('</code>', 'self.end_code_paragraph'),
            ('<delimeter>', 'self.insert_delimeter'))

        for line in self.sourcelines:
            foundtag = 0
            for tag,function in tags:
                if line.lower().startswith(tag):
                    exec function + '(\'' + line.rstrip() + '\')'
                    foundtag = 1
            if not foundtag:
                self.parse_text_line(line)


    def get_filename(self, line):
        self.pyfile = line.rstrip()[6:]

    def title(self, line):
        try:
            title = self.pyfile
        except:
            title = self.sourcefile[:-4]
        self.file.write('<TITLE>\nReference: ' + title +\
                        '\n</TITLE>\n\n<BODY bgcolor="#ffffff" ' +\
                        'text="#000000">\n\n')

        pagelabel = line.rstrip()[7:]
        self.file.write('<CENTER>\n<H2>' + pagelabel + '<BR>\n')
        width = str(40 + len(pagelabel)*10)
        self.file.write('<IMG src="pics/blue.gif" width=' + width +\
                        ' height=2></H2>\n</CENTER>\n\n')

        try:
            self.file.write('<DL>\n\n<DT><H4>File</H4></DT>\n' +\
                            '<DD><A href="../' + self.pyfile + '">' +\
                            self.pyfile + '</A></DD>\n\n')
        except:
            self.file.write('<DL>\n\n')

    def modules_list(self, line):
        file = open('../' + self.pyfile, 'r')
        lines = file.readlines()
        file.close()
        self.file.write('\n<DT><H4>Modules</H4></DT>\n<DD>\n')
        for line in lines:
            if re.search('import\s+', line):
                self.file.write('<CODE>' + line.rstrip() + '</CODE><BR>\n')
        self.file.write('</DD>\n\n')

    def contents_list(self, line):
        self.file.write('\n<DT><H4>Contents</H4></DT>\n' +\
                        '<UL type="disc">\n')
        for line in self.sourcelines:
            if line.lower().startswith('<section>'):
                name = line.rstrip()[9:]
```

```
            self.file.write(' <LI><A href="#' + name + '">' +\
                            name + '</A>\n')
        self.file.write('</UL>\n')

    def start_section(self, line):
        name = line.rstrip()[9:]
        self.file.write('\n\n<P>\n<DT>\n<HR><A name="' + name + '">')
        self.file.write('<H4>' + name + '</H4></A></DT>\n<DD>\n')
        self.currentSectionName = name

    def end_section(self, line):
        # Check if any code is to be inserted
        if re.search('code', line, re.I):
            code = find_code('../' + self.pyfile, self.currentSectionName)
            self.insert_delimeter()
            self.file.write('<FONT color="#000000"><PRE>\n')
            for line in code:
                line = self.fix_symbols(line)
                self.file.write(line + '\n')
            self.file.write('</PRE></FONT>\n')
            self.insert_delimeter()

        elif re.search('file', line, re.I):
            self.file.write('\n<P><FONT color="#000000"><PRE>\n')
            file = open('../' + self.pyfile, 'r')
            lines = file.readlines()
            file.close()
            for line in lines:
                line = self.fix_symbols(line)
                self.file.write(line)
            self.file.write('</PRE></FONT>\n')
            self.insert_delimeter()

        # End section
        self.file.write('</DD>\n')

    def start_code_paragraph(self, line):
        self.file.write('<DL>\n<DD><FONT color="#0033bb"><PRE>\n')
        self.codeOn = 1

    def end_code_paragraph(self, line):
        self.file.write('</PRE></FONT></DD>\n</DL>\n')
        self.codeOn = 0

    def insert_delimeter(self, line = None):
        imagename = 'pics/delimeter.gif'
        self.file.write('\n<P>\n<CENTER><IMG src="' + imagename + '">' +\
                        '</CENTER>\n')

    def parse_text_line(self, line):
        if (line == '\n' and not self.codeOn):
            if not self.noNewParagraph:
                self.file.write('<P>\n')
                self.noNewParagraph = 1
        else:
            if self.codeOn:
                line = self.fix_symbols(line)
                self.file.write(line)
            else:
```

```
                texts = re.split('(<.+?>)', line)
                for text in texts:
                    if text.lower() == '<func>':
                        self.funcOn = 1
                    elif text.lower() == '</func>':
                        self.funcOn = 0
                    else:
                        if self.funcOn:
                            # name = function name + "(" -> bold
                            # args = function arguments  -> italic
                            # end = ")"                  -> bold
                            search = re.search('(.+?\()(.*)(\))', text)
                            name,args,end = search.groups()
                            self.file.write('<B>' + name + '</B>' +\
                                            '<I>' + args + '</I>' +\
                                            '<B>' + end + '</B>')
                        else:
                            self.file.write(text)
                self.noNewParagraph = 0

    def fix_symbols(self, text):
        text = re.sub('<', '&lt;', text)
        text = re.sub('>', '&gt;', text)
        return text

    def close_html_file(self):
        text = '\n</DL>\n\n' +\
               '<CENTER>\n<I><A href="index.html">index</A><BR>\n' +\
               'Morten Wang Fagerland, 2003<BR>\n' +\
               '(mortenfa at ifi.uio.no)</I><BR>\n' +\
               '<IMG src="pics/blue.gif" width=200 height=2>\n' +\
               '</CENTER>\n\n</BODY>\n</HTML>\n'
        self.file.write(text)
        self.file.close()


# Find a class or a function with name <classOrFunction> in <filename>
def find_code(filename, classOrFunction):
    file = open(filename, 'r')
    lines = file.readlines()
    file.close()

    code = []
    found = 0
    for line in lines:
        if re.match('(class|def)\s+' + classOrFunction + '(\(|:)', line.rstrip()):
            found = 1
        elif (re.match('\w', line) and not line.startswith('#')):
            found = 0
        if found:
            code.append(line.rstrip())

    # Remove trailing empty lines
    if not code:
        return []
    line = code[-1]
    while not line:
        code = code[:-1]
        line = code[-1]
```

```
    return code


#-------------------------
# Start creating html files
#-------------------------

for filename in files:
    CreateHTML(filename)
```

Part II

Modules

Chapter 5

Coordinate system conversions (coords)

5.1 Physical2CanvasSystem

When you create a model of a physical system with drawings in a *Tkinter.Canvas* you have to convert your original physical coordinates into coordinates suitable to be represented in the canvas. The *canvas coordinate system* have a positive x-axis to the right, a positive y-axis downwards, and the upper left corner as the origin. If we take into consideration that we do not always want to use the whole canvas, we have the situation illustrated in figure 5.1. The values `x0, x1, y0` and `y1` are the borders of the physical coordinate system, while the values `width, height, topMargin` and `leftMargin` are given in canvas units. Looking at the figure it is obvious that the width and the height of the canvas have to be greater than `leftMargin + width` and `topMargin + height` respectively.

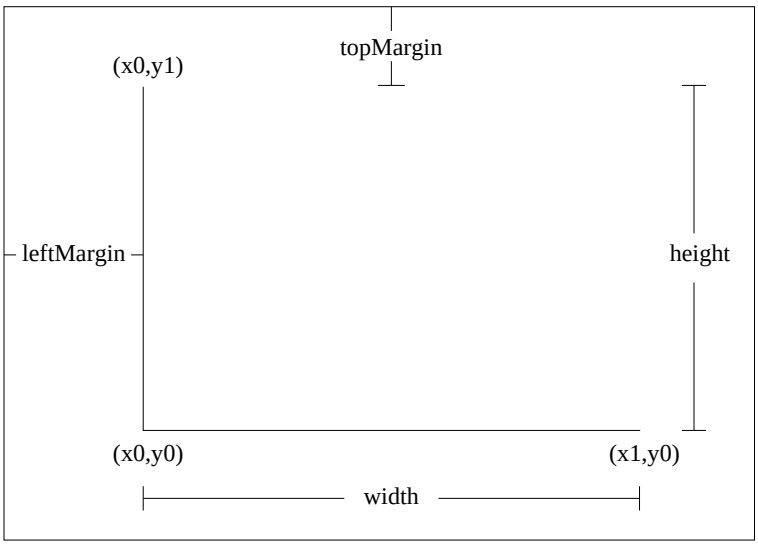


Figure 5.1: The arguments to the class `CoordinateSystem`.

The class `Physical2CanvasSystem` manages the most common coordinate conver-

sions between physical coordinates and canvas coordinates. It is initialised by six arguments (`x0, x1, y0, y1, width` and `height`) and two optional keyword arguments (`topMargin` and `leftMargin`). The keywords default to zero.

`Physical2CanvasSystem` has four functions:

- `coord2canvas(x,y)` calculates and returns canvas coordinates (`cx,cy`) for the given physical coordinates (`x,y`).
- `canvas2coord(cx,cy)` calculates and returns physical coordinates (`x,y`) for the given canvas coordinates (`cx,cy`).
- `coord_length2canvas_length(dx,dy)` calculates and returns canvas lengths (`cdx, cdy`) for the given physical lengths (`dx,dy`).
- `canvas_length2coord_length(cdx,cdy)` calculates and returns physcial lengths (`dx,dy`) for the given canvas lengths (`cdx,cdy`).

```
class Physical2CanvasSystem:
    """Common 2D physical coordinates <=> canvas coordinates conversions"""

    def __init__(self, x0,x1, y0,y1, width,height,
                 leftMargin = 0,
                 topMargin = 0):

        self.x0 = float(x0)           # x_min (physical coordinates)
        self.x1 = float(x1)           # x_max (physical coordinates)
        self.y0 = float(y0)           # y_min (physical coordinates)
        self.y1 = float(y1)           # y_max (physical coordinates)
        self.width = width            # Width of canvas (usable width)
        self.height = height          # Height of canvas (usable height)
        self.leftMargin = leftMargin  # Optional space at the left side
        self.topMargin = topMargin    # Optional space at the top

    # Calculate canvas coordinates (cx,cy) given physical coordinates (x,y)
    def coord2canvas(self, x, y):
        cx = self.leftMargin + self.width*(x-self.x0)/(self.x1-self.x0)
        fraction = (y-self.y0)/(self.y1-self.y0)
        cy = self.topMargin + self.height - self.height*fraction
        return cx,cy

    # Calculate physical coordinates (x,y) given canvas coordinates (cx,cy)
    def canvas2coord(self, cx, cy):
        x = self.x0 + (cx-self.leftMargin)*(self.x1-self.x0)/self.width
        fraction = (self.y1-self.y0)/self.height
        y = self.y0 + (self.height-(cy-self.topMargin))*fraction
        return x,y

    # Calculate canvas lengths (cdx,cdy) given physical lengths (dx,dy)
    def coord_length2canvas_length(self, dx, dy):
        cdx = self.width*dx/(self.x1-self.x0)
        cdy = self.height*dy/(self.y1-self.y0)
        return cdx,cdy

    # Calculate physical lengths (dx,dy) given canvas lengths (cdx,cdy)
    def canvas_length2coord_length(self, cdx, cdy):
        dx = cdx*(self.x1-self.x0)/self.width
```

```
        dy = cdy*(self.y1-self.y0)/self.height
        return dx,dy
```

5.2 cart2polar

```
from math import *

def cart2polar(x, y):
    """Calculate and return the polar coordinates (r, theta)
       given the cartesian coordinates (x, y). theta will be
       expressed in the interval [0, 2*pi)."""

    r = sqrt(x*x + y*y)
    if x == 0:
        if y == 0:
            theta = 0.0
        elif y > 0:
            theta = pi/2
        else:
            theta = 3*pi/2
    elif x > 0:
        if y >= 0:
            theta = atan(float(y)/x)
        else:
            theta = 2*pi + atan(float(y)/x)
    else:
        theta = pi + atan(float(y)/x)
    return r,theta
```

5.3 polar2cart

```
from math import *

def polar2cart(r, theta):
    """Calculate and return the cartesian coordinates (x, y)
    given the polar coordinates (r, theta)."""

    x = r*cos(theta)
    y = r*sin(theta)
    return x,y
```

Chapter 6

Common dialogs (`dialogs`)

6.1 BasePrefs

This class can be used to create a standard preferences toplevel. The toplevel may contain the widget types *counter*, *radioselect* and *entryfield*. It is used by creating a subclass of `BasePrefs` like this:

```
from modules import dialogs

class Prefs(dialogs.BasePrefs):
    def __init__(self, balloon = None):
        dialogs.BasePrefs.__init__(self, balloon)

        # Create the preference values as class attributes here:
        self.a = 1.0
        self.choice = 'This'
        [...]

    def launch_window(self):
        self.create_dialog(root, width = 240, height = 200)

        # All counters may be specified as a list of this type:
        # (label, variable, type, min, max, balloon_text)
        list = (('The value of variable a: ', self.a, 'real', 0.0, 5.0,
                 'a is a real number (0.0 - 5.0)'),)
        [...]

        # Other widget types may be create as this radioselect
        # (label, buttons, pady)
        args = ('Choose one: ', ('This', 'Or this', 'Maybe this'), 10)
        self.radioSelect = self.create_radioselect(*args)
        self.radioSelect.invoke(self.choice)

    # This function is called whenever a button in the dialog is pressed
    def close_window(self, button):
        if button == 'OK':
```

```
                self.a = float(self.counters[0].get())
                self.choice = self.radioSelect.getcurselection()
            self.dialog.destroy()
```

The BasePrefs demonstration (section A.2) is a more comprehensive example of how to use BasePrefs. Another example using the display_error_message function may be found in the matrix application (section 13).

```
import Tkinter
import Pmw
from modules import misc

class BasePrefs:
    """A set of convenience functions used to create common widgets
       in a separate preferences dialog"""

    def __init__(self, balloon = None):
        self.balloon = balloon

    # Create, process and return a Pmw.Dialog
    def create_dialog(self, root, width = 240, height = 390,
                      buttons = ('Cancel', 'OK'), defaultbutton = 1):
        self.dialog = Pmw.Dialog(root,
                buttons = buttons,
                command = self.close_window,
                defaultbutton = defaultbutton,
                title = 'Preferences')
        misc.process_toplevel(self.dialog, root, width, height)

    # Create an error message in a separate dialog
    def display_error_message(self, message, width, height):
        top = Pmw.MessageDialog(self.dialog.interior(),
                borderx = 20,
                bordery = 5,
                defaultbutton = 0,
                iconmargin = 0,
                iconpos = 'n',
                icon_bitmap = 'error',
                message_text = message,
                title = 'Warning!')
        misc.process_toplevel(top, self.dialog, width, height)

    # Create a list of Pmw.Counters from the options in list
    def create_counters(self, list):
        counters = []
        for args in list:
            counter = self.create_counter(*args)
            counters.append(counter)
        Pmw.alignlabels(counters)
        return counters

    # Create, pack and return a Pmw.Counter
    def create_counter(self, label, value, type, min, max, balloon):
        if type == 'real':
            increment = .1
        else:
            increment = 1
```

```
        validator = {'validator': type, 'min': min, 'max': max}
        counter = Pmw.Counter(self.dialog.interior(),
                datatype = type,
                entryfield_validate = validator,
                entryfield_value = value,
                entry_justify = 'center',
                entry_width = 5,
                increment = increment,
                labelpos = 'w',
                label_text = label)
        counter.pack(padx = 10, pady = 5)
        if self.balloon:
            self.balloon.bind(counter, balloon)
        return counter

    # Create, pack and return a Pmw.RadioSelect
    def create_radioselect(self, text, buttons, pady):
        radioSelect = Pmw.RadioSelect(self.dialog.interior(),
                buttontype = 'radiobutton',
                hull_borderwidth = 2,
                hull_relief = 'groove',
                labelpos = 'n',
                label_text = text)
        radioSelect.pack(padx = 10, pady = pady)
        for button in buttons:
            radioSelect.add(button)
        return radioSelect

    # Create, pack and return a Pmw.EntryField
    def create_entryfield(self, text, width, value, pady):
        entryField = Pmw.EntryField(self.dialog.interior(),
                entry_width = width,
                labelpos = 'w',
                label_text = text,
                value = value)
        entryField.pack(padx = 10, pady = pady)
        return entryField
```

Chapter 7

Function and parametric curve properties (`functions`)

7.1 Function

In several mathematical models we find equations of the type $y = f(x)$ that has to be solved for either some discrete values of x or on some given interval. The class `Function` takes care of some basic operations for arbitrary functions. `Function` requires three arguments: the function written as a string (for example "`sin(x)+.5*x^2`") and the minimum and maximum of the domain of the function (i.e. the interval $[x0, x1]$). The optional keyword arguments `variable` and `samples` can be used to change the name of the independent variable and the number of calculated values in the interval.

`Function` has the following functions:

- `calculate_value(x)` returns the value $f(x)$ for a given `x`.
- `calculate_values()` returns a list of `samples` evenly spaced values $\big(x, f(x)\big)$ from the interval `[x0, x1]`.
- `min(interval = 'all')` returns the minimum value of `calculated_values()`. You may use another range of x values, other than the interval defined by the class attributes `x0` and `x1`, by supplying a sequence of two elements to the `interval` keyword (see the example below).
- `max(interval = 'all')` returns the maximum value of `calculated_values()`. You may use another range of x values, other than the interval defined by the class attributes `x0` and `x1`, by supplying a sequence of two elements to the `interval` keyword (see the example below).

Example:

```
from modules import functions
from math import*

function = functions.Function('sin(x)', 0, 2*pi, samples = 20)
for x,y in function.calculate_values():
    print 'x: %(x)6.6f   f(x): %(y)6.6f' %vars()
```

```
    print 'min:', function.min()
    print 'max:', function.max(interval = [0, pi/4])

import re
from math import *

class Function:
    """Basic properties of a function y=f(x)"""

    def __init__(self, function, x0, x1,
                 samples = 100,
                 variable = 'x'):

        self.function = function  # The function f(x) as a text string
        self.x0 = x0              # Define the interval [x0,x1]
        self.x1 = x1              # Define the interval [x0,x1]
        self.samples = samples    # Number of values to calculate from [x0,x1]
        self.variable = variable  # The name of the independent variable

    # Let class instances be callable and return value y=f(x)
    def __call__(self, x):
        return self.calculate_value(x)

    # Return the value y=f(x)
    def calculate_value(self, x):
        return eval(re.sub(self.variable, str(x), self.function))

    # Return samples evenly spaced values (x,y=f(x)) from [x0,x1]
    def calculate_values(self):
        values = []
        dx = (self.x1-self.x0)/float(self.samples-1)
        for i in range(self.samples):
            x = self.x0+i*dx
            values.append((x, self.calculate_value(x)))
        return values

    # Return the minimum value for y from calculate_values()
    def min(self, interval = 'all'):
        if interval == 'all':
            x0 = self.x0
            x1 = self.x1
        else:
            x0,x1 = map(float, interval)
        values = []
        for x,y in self.calculate_values():
            if (x >= x0 and x <= x1):
                values.append(y)
        return min(values)

    # Return the maximum value for y from calculate_values()
    def max(self, interval = 'all'):
        if interval == 'all':
            x0 = self.x0
            x1 = self.x1
        else:
            x0,x1 = map(float, interval)
        values = []
        for x,y in self.calculate_values():
            if (x >= x0 and x <= x1):
```

```
                values.append(y)
        return max(values)
```

7.2 ParametricCurve

This class is a natural extension of the class `Function` to parametric curves. A parametric curve is defined as the set $\{(x(t), y(t)) \mid t \in [t_0, t_1]\}$ for the functions $x(t)$ and $y(t)$ on some interval $[t_0, t_1]$. `ParametricCurve` has almost the same user interface as `Function`. The only difference when you initialise the class is that you have to have two strings to represent the functions $x(t)$ and $y(t)$.

The functions offered by this class are analogous to `Function`'s.

- `calculate_value(t)` returns the pair $\big(x(t), y(t)\big)$ for a given `t`.
- `calculate_values()` returns a list of `samples` evenly spaced values $\big(t, x(t), y(t)\big)$ from the interval `[t0, t1]`.
- `min(interval = 'all')` returns the minimum values of `calculated_values()`. You may use another range of t values, other than the interval defined by the class attributes `t0` and `t1`, by supplying a sequence of two elements to the `interval` keyword. For example:

  ```
  function = functions.ParametricCurve('cos(t)', 'sin(t)', 0, 2*pi)
  minx,miny = function.min([0, pi])
  ```

- `max(interval = 'all')` returns the maximum values of `calculated_values()`. You may use another range of t values, other than the interval defined by the class attributes `t0` and `t1`, by supplying a sequence of two elements to the `interval` keyword.

```
import re
from math import *

class ParametricCurve:
    """Basic properties of a parametric curve (x(t),y(t))"""

    def __init__(self, func_x, func_y, t0, t1,
                 samples = 100,
                 variable = 't'):

        self.func_x = func_x      # The function x(t) as a text string
        self.func_y = func_y      # The function y(t) as a text string
        self.t0 = t0              # Define the interval [t0,t1]
        self.t1 = t1              # Define the interval [t0,t1]
        self.samples = samples    # Number of values to calculate from [t0,t1]
        self.variable = variable  # The name of the independent variable

    # Let class instances be callable and return value (x(t),y(t))
    def __call__(self, t):
        return self.calculate_value(t)

    # Return the value (x(t),y(t))
```

```
def calculate_value(self, t):
    x = eval(re.sub(self.variable, str(t), self.func_x))
    y = eval(re.sub(self.variable, str(t), self.func_y))
    return x,y

# Return samples evenly spaced values (t,x(t),y(t)) from [t0,t1]
def calculate_values(self):
    values = []
    dt = (self.t1-self.t0)/float(self.samples-1)
    for i in range(self.samples):
        t = self.t0+i*dt
        x,y = self.calculate_value(t)
        values.append((t,x,y))
    return values

# Return the minimum value for x and y from calculate_values()
def min(self, interval = 'all'):
    if interval == 'all':
        t0 = self.t0
        t1 = self.t1
    else:
        t0,t1 = map(float, interval)
    values_x = []
    values_y = []
    for t,x,y in self.calculate_values():
        if (t >= t0 and t <= t1):
            values_x.append(x)
            values_y.append(y)
    return min(values_x),min(values_y)

# Return the maximum value for x and y from calculate_values()
def max(self, interval = 'all'):
    if interval == 'all':
        t0 = self.t0
        t1 = self.t1
    else:
        t0,t1 = map(float, interval)
    values_x = []
    values_y = []
    for t,x,y in self.calculate_values():
        if (t >= t0 and t <= t1):
            values_x.append(x)
            values_y.append(y)
    return max(values_x),max(values_y)
```

Chapter 8

Miscelaneous classes and functions (misc)

8.1 Command

This class is used to avoid the somewhat cumbersome syntax of lambda expressions. Lambda expressions are used to delay the call to a function, usually in connection with a `bind` statement. The `Command` class was first published by Timothy R. Evans on a Python newsgroup.

To bind the left mouse button in a widget (say a *Tkinter.Canvas*) to the function `LMB_pressed(event, p1, p2)`, `Command` may be used like this:

```
canvas.bind('<Button-1>', Command(LMB_pressed, p1, p2))
```

The equivalent lambda expression code:

```
canvas.bind('<Button-1>', lambda event=None, p1=p1, p2=p2,
    LMB=LMB_pressed: LMB(event, p1, p2))
```

The advantage is a simpler syntax, and easy-to-read, easy-to-maintain code.

```
class Command:
    """Alternative to lambda functions
       by Timothy R. Evans"""

    def __init__(self, func, *args, **kw):
        self.func = func
        self.args = args
        self.kw = kw

    def __call__(self, *args, **kw):
        args = args + self.args
        kw.update(self.kw)  # override kw with orig self.kw
        apply(self.func, args, kw)
```

8.2 MenuBarConfig

If you like your menubar buttons to highlight when the mouse cursor moves over them, this class does the trick. It will also configure the vertical padding (the `pady` option) and the borderwidth (the `bd` option) of each button. Use it like this:

```
menubar = Pmw.MenuBar(master)
menubar.addmenu('File', '')
menubar.addmenu('Options', '')
menubar.addmenu('Miscelaneous', '')
[...]
buttons = ('File', 'Options', 'Miscelaneous')
MenuBarConfig(menubar).configure(buttons)
```

```
class MenuBarConfig:
    """Configure menubar; bind <Enter> and <Leave> events
       and resize menu buttons"""

    def __init__(self, menubar):
        self.menubar = menubar

    # Define bindings and resize buttons
    def configure(self, buttons, pady = 2, bd = 1):
        for button in buttons:
            component = button + '-button'
            self.menubar.component(component).configure(pady = pady, bd = bd)

            args = ('<Enter>', Command(self._entered_menu, component))
            self.menubar.component(component).bind(*args)

            args = ('<Leave>', Command(self._left_menu, component))
            self.menubar.component(component).bind(*args)

    # Raise menu button
    def _entered_menu(self, event, button):
        self.menubar.component(button).configure(relief = 'raised')

    # Flatten menu button
    def _left_menu(self, event, button):
        self.menubar.component(button).configure(relief = 'flat')
```

8.3 calculate_geometry

This function is very useful in applications with one master window and smaller popup windows. It is best illustrated with an example:

```
master = Tkinter.Tk()
[...]
top = Tkinter.Toplevel()
top.geometry(calculate_geometry(master, 200, 150))
```

The width and height of the new window (`top`) will be 200 and 150 respectively, and it will be positioned at the center of the master window. You may add two more options (`xoffset` and `yoffset`) to move the window some distance from the center.

```
def calculate_geometry(master, width, height, xoffset = 0, yoffset = 0):
    """Return a centered (+xoffset, +yoffset)
       geometry value wrt a master window"""

    size,x,y = master.geometry().split('+')
    dx0,dy0 = map(int, size.split('x'))
```

```
xpos = -(width/2) + int(x) + xoffset + dx0/2
ypos = -(height/2) + int(y) + yoffset + dy0/2

return '%ix%i+%i+%i' % (width, height, xpos, ypos)
```

8.4 create_path

It is often necessary to move a *Tkinter.Canvas* object from one position in the canvas to another. The object is supposed to move the shortest possible distance (a straight line), at least one canvas unit along the direction of at least one of the coordinate axes, but no more than one unit in each direction in each iteration. I.e. an object may move one unit in the north/south direction and/or one unit in the east/west direction, but not two (or more) units in one direction or no units in any direction.

Let (dx, dy) be the number of units an object may move in each direction in each iteration. We then have eight posibilities (see figure 8.1): (1, 1), (1, 0), (1, -1), (0, 1), (0, -1), (-1, 1), (-1, 0), (-1, 1).

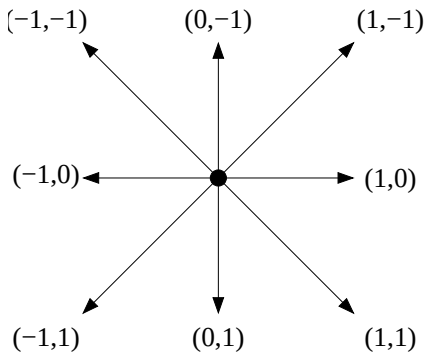


Figure 8.1: The allowable values of (dx, dy) in each iteration.

The function expects the coordinates of the start and end position of the object. A list of canvas units `[(dx, dy), (dx, dy), ...]` will be returned. This list may be used in combination with the *Tkinter.Canvas* function `move` to move the object each iteration (see the example further down the page).

The idea behind the algorithm is as follows:

- Let dx and dy be the total distance the object is supposed to move to the right and downwards respectively.
- Find the greatest value of dx and dy. The length of the list to return, and also the number of iterations, will equal this value. Let us now assume that $dx > dy$.
- Move the object `sign(dx)` units to the right in each iteration.
- Move the object `sign(dy)` units downwards for every dy/dx'th iteration.

Example:

The code needed to move an object with tag `item` from canvas coordinates `(x0, y0)` to coordinates `(x1, y1)`:

```
[...]
for dx,dy in create_path(x0,y0, x1,y1):
canvas.move(item, dx, dy)
canvas.update()
[...]
```

Example:

If $(x0, y0) = (0, 4)$ and $(x1, y1) = (8, 0)$, $dx = 8$ and $dy = -4$. `create_path` will then return the list:

$$\big[(1,-1),(1,0),(1,-1),(1,0),(1,-1),(1,0),(1,-1),(1,0)\big].$$

This is not of course a straight line, the object will move a distance of either one canvas unit or $\sqrt{2}$ canvas units each iteration, i.e. not a 100% smooth movement, but a pretty good approximation.

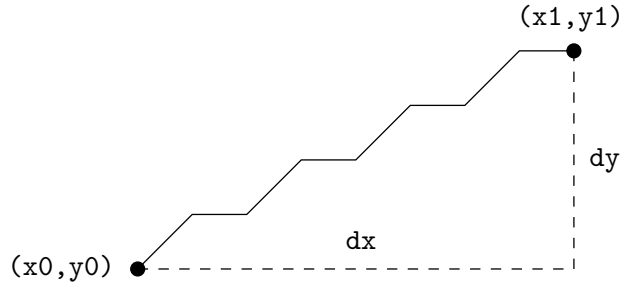


Figure 8.2: An `create_path` example with $dx = 8$ and $dy = -4$.

```
def create_path(x0,y0, x1,y1):
    """Calculates a path from x0,y0 to x1,y1 and returns the path as a list"""

    dx = x1 - x0    # Distance to move in the x-direction
    dy = y1 - y0    # Distance to move in the y-direction
    path = []       # A list of elements (dx,dy)
    n = 0           # Number of non-zero entries in path

    if not (dx or dy):
        return []

    for i in range(max((abs(dx),abs(dy)))):
        if abs(dx) > abs(dy):
            x_step = sign(dx)
            if n <= i*abs(float(dy)/dx):
                y_step = sign(dy)
                n = n + 1
            else:
                y_step = 0
```

```
        else:
            y_step = sign(dy)
            if n <= i*abs(float(dx)/dy):
                x_step = sign(dx)
                n = n + 1
            else:
                x_step = 0
        path.append((x_step, y_step))
    return path
```

8.5 process_toplevel

The same configurations usually need to be applied to all *Tkinter.Toplevels*. This includes centering the toplevel with respect to a master window and giving the new toplevel focus. Sometimes I want to disable resizing and make the toplevel transient. This function will save me some time when I'm programming and make sure that all my windows have the same functionality. It will shorten the code a little bit, too. Note that it uses the function `calculate_geometry` described above.

```
def process_toplevel(top, master, width, height, xoffset = 0, yoffset = 0,
                     resizable = (0,0), transient = 1, focus = 1):
    """Process toplevel window; set size, position, etc"""

    top.geometry(calculate_geometry(master, width, height, xoffset, yoffset))
    top.resizable(resizable[0], resizable[1])
    if focus:
        top.focus()
    if transient:
        top.transient(master)
```

8.6 rotate_point

This function rotates a point `(x, y)` a given angle `angle` with respect to another point `(x0, y0)` (see figure 8.3). As rotation is most easily described in polar coordinates, the function starts by defining the point `(x, y)` in polar coordinates `(r, theta)` with `(x0, y0)` as the origin, using the `cart2polar` function from the `coords` module (section 5.3). After subtracting `angle` from `theta`, the point is redefined in carthesian coordinates. A positive angle has an anticlockwise direction.

```
from modules import coords
from math import *

def rotate_point(x,y, x0,y0, angle):
    """Rotate a point (x,y) wrt (x0,y0) and return new coordinates"""

    # Find the distance between the points
    dx = x - x0
    dy = y - y0

    # Define the point (relative to (x0,y0)) in polar coordinates (r,theta)
    r,theta = coords.cart2polar(dx, dy)

    # Add angle and compute new coordinates
    theta = theta - angle
```

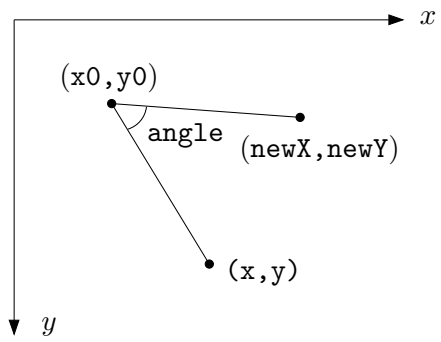


Figure 8.3: Rotation of the point (x, y) with respect to (x_0, y_0).

```
    dx = r*cos(theta)
    dy = r*sin(theta)
    return x0+dx,y0+dy
```

8.7 sign

```
def sign(number):
    """Return the sign of the given number"""

    if number > 0:
        return 1
    if number == 0:
        return 0
    if number < 0:
        return -1
```

Chapter 9

Handle algorithm flow/animation (programflow)

9.1 ProgramFlow

This class is very useful to keep control of the program flow of an algorithm. It may be used to run through the entire algorithm at once (animation), pause it, restart the algorithm, or use a step by step method. It is initialised with a master window (for example the root window) and the function that acts as the algorithm:

```
from modules import programflow

class Gui:
    def __init__(self):
        self.flow = programflow.ProgramFlow(root, self.algorithm)

    def algorithm(self):
        [...]

root = Tkinter.Tk()
Gui()
```

Use the function `wait_or_next_step(speed = 1, seconds = 0)` every time you want a pause in the algorithm. It takes two optional keyword arguments to control the length of the pause: `speed` and `seconds`. `speed` may be used to implement variable pause length; the greater the value the longer the pause (a `speed` of 1 equals no pause). The effect of increasing the `speed` value by one may be changed with the `speedFactor` argument. A value of `seconds` means, of course, a pause given in seconds. The pauses are ignored if you use the step by step method. The program will then halt the execution of the algorithm and wait for the next step to be invoked.

Example:

A simple algorithm that prints the numbers 1 - 20 and takes a pause after each number has been printed:

```
def algorithm(self):
```

```
        for i in range(20):
            print i + 1
            self.flow.wait_or_next_step(seconds = .2)
        self.flow.animation = 0
        self.flow.started = 0
```

The last two lines let the `ProgramFlow` class know that the algorithm has ended. The animation may be started by the function `start_animation()`. If the algorithm has already started (as indicated by the `started` attribute) this will not restart the algorithm but instead animate the rest of it. If the algorithm is not in animation mode (indicated by the `animation` attribute) the function `next_step()` will execute the next step in the algorithm (i.e. up until the next call to `wait_or_next_step`).

You may restart the algorithm at any time with a call to `restart()`. If the program is in animation mode, the animation starts all over at once. A call to `stop_animation()` will halt the animation until the `start_animation` function is called. It is advisable to disable calls to `stop_animation` when not in animation mode as this may confuse the program.

Hopefully the `ProgramFlow` demonstration (section A.4) will clearify matters.

```
import Tkinter
import time

class ProgramFlow:
    def __init__(self, master, algorithm, speedFactor = 3000):
        self.master = master
        self.algorithm = algorithm
        self.speedFactor = speedFactor

        # Animation or step-by-step?
        self.animation = 0

        # Has the algorithm started?
        self.started = 0

        # Used to halt the animation in the wait_or_next_step function
        self.nextStepVar = Tkinter.IntVar()

    # Restart the algorithm
    def restart(self):
        self.started = 0
        if self.animation:
            self.next_step()

    # Start (or resume) the animation of the (rest of the) algorithm
    def start_animation(self):
        self.animation = 1
        self.next_step()

    # Stop (or pause) the animation
    def stop_animation(self):
        self.animation = 0
        self.nextStepVar.set(1)

    # Go to the next step in the algorithm.
    # If this is the first step: initialise the algorithm
    def next_step(self):
```

```
        if self.started:
            self.nextStepVar.set(1)
        else:
            self.started = 1
            if callable(self.algorithm):
                self.algorithm()

    # If animation: update and pause
    # If step by step: halt algorithm execution
    def wait_or_next_step(self, speed = 1, seconds = 0):
        if self.animation:
            self.update_and_pause(speed, seconds)
        else:
            self.master.wait_variable(self.nextStepVar)

    # Update master window and take a pause.
    def update_and_pause(self, speed = 1, seconds = 0):
        self.master.update()
        for i in range((speed-1)*self.speedFactor):
            pass
        time.sleep(seconds)
```

Chapter 10

New widgets (`widgets`)

10.1 Spring

The spring mass system is a popular model in physics and applied mathematics. To create an illustration involving such a system, some way to draw the spring had to be found. Two different designs leapt to mind. The most suitable was found by testing their visual attractiveness in a head-to-head contest (section A.7).

The `Spring` class makes it easy to draw a string in a *Tkinter.Canvas* and to dynamically adjust its length. Initialise it like this:

```
from modules import widgets

x = 100
y = 100
length = 150
displacement = 0.0

# Assume we have a Tkinter.Canvas called "canvas"
spring = widgets.Spring(canvas, x, y, length)
spring.draw(displacement)
```

The `x` and `y` options are the position of the spring. If the `angle` (see below) is zero (default) this corresponds to the top center of the spring. The `length` option is the total length of the spring (in canvas units) in its equilibrium state.

The `spring.draw(displacement)` function draws the spring while the given option `displacement` is a value in the range $[-1.0, 1.0]$ indicating the spring's displacement from equilibrium. The values -1.0 and 1.0 correspond to the spring's maximum compressed state and it's maximum outstretched state respectively. A warning message will be printed if these limits are exceeded, but the program will not stop. This will probably happen frequently if you run the forced vibrations application (section 16.5).

`Spring` has several optional keyword arguments that can be used to modify the appearance of the spring:

- `angle = 0`: The spring doesn't have to be positioned from top to bottom; it may have an arbitrary alignment. Indicate the direction by stating the angle between the spring and the y-axis. A positive angle has an anticlockwise direction.

- `fill = 'Black'`: The colour of the spring.
- `noOfLoops = 5`: The number of loops in the spring.
- `samples = 100`: The number of points in the parametric curve to be computed (see the `Spring_math` class below). A linear interpolation connects each point. (There may be other (better) methods to describe the spring; perhaps with splines. They are not considered here.)
- `startEndLength = 'use standard ratio'`: The first and last straight line of the spring (see figure 10.1) may be given an explicit length (in canvas units). If no value is given the lines will have length according to

$$\texttt{startEndLength} = \frac{12}{130}\texttt{springLength}$$

- `tags = ['spring',]`: A list of tags.
- `width = 'use standard ratio'`: The width of the spring may be given an explicit length (in canvas units). If no value is given the width will be set to

$$\texttt{width} = \frac{18}{130}\texttt{springLength}$$

See the `Spring` demonstration (section A.6) for an example of these features.

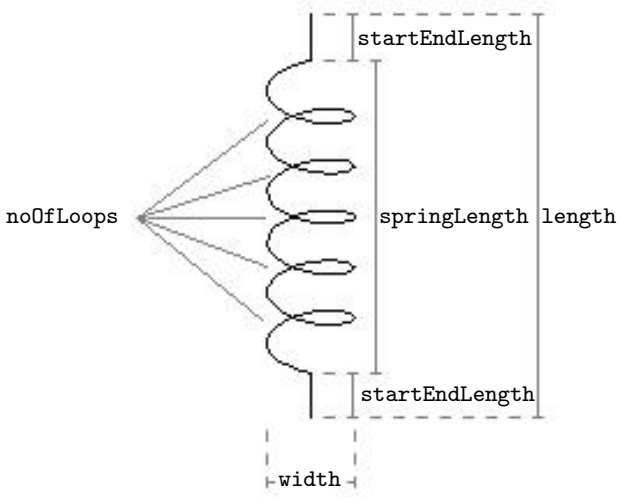


Figure 10.1: Some of the options that control the spring's appearance.

In addition to the `draw` function described above, `Spring` has two more global functions:

- `get_total_length(displacement)` returns the total length of the spring with the given displacement. Note that this value will be the same as the initial argument `length` when the spring is in equilibrium (`displacement = 0`).
- `get_end_position(displacement)` returns the canvas coordinates (x, y) corresponding to the spring's end position with the given displacement. The `rotate_point` function from the `misc` module (section 8) is used.

```
from modules import misc

class Spring(Spring_math):
    """Defines a spring to be used in a Tkinter.Canvas"""

    def __init__(self, canvas, x, y, length,
                 angle = 0,
                 fill = 'Black',
                 noOfLoops = 5,
                 samples = 100,
                 startEndLength = 'use standard ratio',
                 tags = ['spring',],
                 width = 'use standard ratio'):

        # Initialize base class
        Spring_math.__init__(self, noOfLoops, samples)

        self.canvas = canvas  # Where to draw the spring
        self.x = x            # Start drawing spring at coordinates (x,y)
        self.y = y            # Start drawing spring at coordinates (x,y)
        self.angle = angle    # Angle from y-axis (counterclockwise)
        self.fill = fill      # The colour of the spring
        self.tags = tags      # The canvas tags

        self._calculate_lengths(length, startEndLength, width)

        # Normalize the springLength
        min,max = self._get_y_min_max(0.0)
        self.springLengthNorm = self.springLength/float(max-min)

    # Set lengths and width. If not specified use standard ratios
    # length = startEndLength + springLength
    # startEndLength = 12*springLength/130  (standard ratio)
    # width = 18*springLength/130           (standard ratio)
    def _calculate_lengths(self, length, startEndLength, width):
        if startEndLength == 'use standard ratio':
            self.springLength = length/float(1+24.0/130)
            self.startEndLength = 12*self.springLength/130.0
        else:
            self.springLength = length-2*startEndLength
            self.startEndLength = startEndLength

        if width == 'use standard ratio':
            self.width = 18*self.springLength/130.0
        else:
            self.width = width

    # Calculate and draw the spring with given displacement
    def draw(self, displacement = 0.0):
        if (displacement < -1.0 or displacement > 1.0):
            print 'Warning! displacement not in the range [-1.0, 1.0]'
#            raise ValueError, 'displacement should be in the range [-1.0, 1.0]'

        values = self._calculate_values(displacement)    # Math values
        minY = values[0][2]                               # The first y-value
        cvalues = []                                      # Canvas values

        # The start position for the spring (after the first straight line)
        x0 = self.x
```

```
        y0 = self.y + self.startEndLength - minY*self.springLengthNorm

        # Calculate the position of each point in the spring
        for t,x,y in values:
            x = (x0+self.width*x)             # Position when angle=0
            y = (y0+self.springLengthNorm*y)  # Position when angle=0
            x,y = misc.rotate_point(x,y, self.x,self.y, self.angle)
            cvalues.extend((x,y))

        # Delete old spring
        self.canvas.delete(self.tags)

        # The first straight line
        x0 = self.x
        y0 = self.y
        x1,y1 = misc.rotate_point(x0,y0+self.startEndLength, x0,y0, self.angle)
        kw = {'fill': self.fill, 'tags': self.tags}
        self.canvas.create_line(x0,y0, x1,y1, **kw)

        # The spring
        self.canvas.create_line(cvalues, **kw)

        # The last straight line
        x0,y0 = cvalues[-2:]    # The last x,y values of the spring
        x1,y1 = misc.rotate_point(x0,y0+self.startEndLength, x0,y0, self.angle)
        self.canvas.create_line(x0,y0, x1,y1, **kw)

    # Return the total length of the spring with given displacement
    def get_total_length(self, displacement):
        min,max = self._get_y_min_max(displacement)
        springLength = (max-min)*self.springLengthNorm
        return 2*self.startEndLength+springLength

    # Return the end position of the spring with given displacement
    def get_end_position(self, displacement):
        x0 = self.x
        y0 = self.y + self.get_total_length(displacement)
        x,y = misc.rotate_point(x0,y0, self.x,self.y, self.angle)
        return x,y
```

10.2 Spring_math

This is the base class for the derived class `Spring` described above. It handles the mathematical calculations of the parametric curve used to define the spring. The attributes and functions of this class are used by the derived class and are not meant to be available as a programming interface.

Let's take a look at how the spring is defined. It has three parts: a straight line, a looping curve and another straight line. The looping curve is given by the simple parametric formula

$$x(t) = -a\cos t, \quad y(t) = bt + c\sin t, \quad t \in [t_0, t_1].$$

The length of the interval $[t_0, t_1]$ decides the number of loops in the spring:

$$t_0 = -\pi/2, \quad t_1 = \pi(2 * \texttt{noOfLoops} + 1/2).$$

The values of a, b and c control the width, the stretching factor, and how much the curve doubles back at the loops. The last value may also be described as the "angle" at which to view the spring. The actual values chosen for a, b, c and Δb were found by experimenting with different values. These values may of course be modified to change the basic abilities of the spring, but the optional keyword arguments of `Spring` should in most cases be sufficient to alter the appearance of the spring.

The local functions `_calculate_values(d)` and `_get_y_min_max(d)` are used to find a list of values $\big(x(t), y(t)\big)$ for dicrete values of t and to find the minimum and maximum value of $y(t)$, respectively. The actual calculations of the curve are done by the class `ParametricCurve` from the `functions` module (section 7).

```
from modules import functions
from math import *

class Spring_math:
    """Base class for the derived class Spring. Handles the mathematical
       calculations of the curve used to define the spring"""

    def __init__(self, noOfLoops, samples):
        self.a = 1.0               # The width of the spring
        self.b_equilibrium = 3.4   # When the spring is in equilibrium
        self.db_max = 2.9          # When the spring hits min/max displacement
        self.c = 7.5               # What "angle" to watch the spring
                                   # (How much the curve should double back)
        self.noOfLoops = noOfLoops # The number of loops in the spring
        self.samples = samples     # Number of points used to draw the curve
        self.t0 = -pi/2            # Define the start parameter

        # Define the end parameter value based on noOfLoops
        self.t1 = (2*noOfLoops+.5)*pi

    # Calculate the points that define the spring with displacement -1<=d<=1
    def _calculate_values(self, d):
        b = self.b_equilibrium+d*float(self.db_max)         # Calculate b-value
        func1 = '-' + str(self.a) + '*cos(t)'               # x(t)=-a*cos(t)
        func2 = str(b) + '*t+' + str(self.c) + '*sin(t)'    # y(t)=b*t+c*sin(t)

        # Define parametric curve and return calculated values
        self.curve = functions.ParametricCurve(func1, func2, self.t0, self.t1,
                samples = self.samples,
                variable = 't')
        return self.curve.calculate_values()

    # Calculate y_min and y_max with displacement -1<=d<=1
    def _get_y_min_max(self, d):
        b = self.b_equilibrium + d*float(self.db_max)   # Calculate b-value
        y_min = b*self.t0 + self.c*sin(self.t0)         # Smallest y-value at t0
        y_max = b*self.t1 + self.c*sin(self.t1)         # Largest y-value at t1
        return y_min,y_max
```

10.3 SmoothRectangle

This class was created because there is no standard *Tkinter.Canvas* widget that can be used to draw a rectangle with smooth corners. The size of the corner curves are set at initialisation. There are three ways to do this (see figure 10.2):

- Use the optional keyword arguments dx and dy:

```
widget = widgets.SmoothRectangle(canvas, 100, 100, 250, 200,
dx = 20, dy = 20)
```

- Use the optional keyword argument r:

```
widget = widgets.SmoothRectangle(canvas, 100, 100, 250, 200,
        r = 15)
```

- Use the default setting: dx and dy are set to one third of the width and one third of the the height of the rectangle:

```
widget = widgets.SmoothRectangle(canvas, 100, 100, 250, 250)
```

The first argument is a *Tkinter.Canvas* object. The next four arguments are the upper left and the lower right coordinates (x_0, y_0) and (x_1, y_1).

The widget is drawn with a call to the `draw()` function:

```
widget.draw()
```

Like the *Tkinter.Canvas* functions `create_rectangle` and `create_oval` the class `SmoothRectangle` offers the optional arguments `outline`, `stipple`, `tags` and `width`. The `fill` argument is the only one not supported.

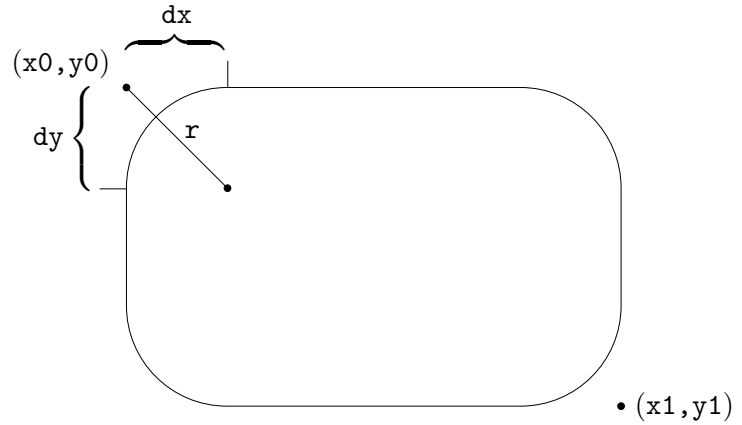


Figure 10.2: How to shape the corners in a `SmoothRectangle`

The actual drawing of the curves are managed by the *Tkinter.Canvas* function `create_line([...], smooth = 1)`. If either `dx = dy = 0` or `r = 0`, `SmoothRectangle` will draw a regular rectangle with squared corners.

See the `SmoothRectangle` demonstration (section A.5) for an example.

```
from math import *

class SmoothRectangle:
    """ Defines a rectangle with smooth corners"""
```

```
def __init__(self, canvas, x0,y0, x1,y1,
             dx = 'use standard ratio',
             dy = 'use standard ratio',
             outline = 'Black',
             r = 'use standard ratio',
             stipple = None,
             tags = None,
             width = 1):

    self.canvas = canvas  # Where to draw the rectangle
    self.x0 = x0          # The left end of the rectangle
    self.y0 = y0          # The top end of the rectangle
    self.x1 = x1          # The right end of the rectangle
    self.y1 = y1          # The bottom end of the rectangle

    # Set the size of the corner curves (by given r or dx/dy)
    # Use the ratio 1/3 of length/height if no value is given
    if r != 'use standard ratio':
        self.dx = r*0.5*sqrt(2)
        self.dy = r*0.5*sqrt(2)
    else:
        if dx == 'use standard ratio':
            self.dx = (self.x1-self.x0)/3.0
        else:
            self.dx = dx
        if dy == 'use standard ratio':
            self.dy = (self.y1-self.y0)/3.0
        else:
            self.dy = dy

    self.outline = outline    # The colour of the rectangle
    self.stipple = stipple    # Stipple brush bitmap name
    self.tags = tags          # Associate rectangle with tags
    self.width = width        # The width of the outline

# Draw the rectangle as eight lines
def draw(self):
    x0, y0 = self.x0, self.y0
    x1, y1 = self.x1, self.y1
    dx, dy = self.dx, self.dy

    lines = (((x0,y0+dy,    x0,y0, x0+dx,y0),   1),     # NW-corner
             ((x0+dx,y0,    x1-dx,y0),          0),     # Top line
             ((x1-dx,y0,    x1,y0, x1,y0+dy),   1),     # NE-corner
             ((x1,y0+dy,    x1,y1-dy),          0),     # Right line
             ((x1,y1-dy,    x1,y1, x1-dx,y1),   1),     # SE-corner
             ((x1-dx,y1,    x0+dx,y1),          0),     # Bottom line
             ((x0+dx,y1,    x0,y1, x0,y1-dy),   1),     # SW-corner
             ((x0,y1-dy,    x0,y0+dy),          0))     # Left line

    self.canvas.delete(self.tags)   # Delete old rectangle

    kw = {'fill': self.outline, 'stipple': self.stipple,
          'width': self.width, 'tags': self.tags}
    for coords,smooth in lines:
        self.canvas.create_line(coords, smooth = smooth, **kw)
```

10.4 create_separator

This function will simply create a thin *Tkinter.Frame* of optional width that can be used as a separator.

```
import Tkinter

def create_separator(master):
    """Create and return a Tkinter.Frame that can be used as a separator"""

    separator = Tkinter.Frame(master,
            borderwidth = 1,
            height = 2,
            relief = 'sunken')
    return separator
```

Chapter 11

Pmw widgets

11.1 Introduction

Python megawidgets (Pmw) is a great collection of high-level compound widgets. It does not, however, cover every need. Luckily, though, it is released with the proper tools and documentation on how to design and code your own megawidgets.

The `MultiListBox` and the `ProgressBarDialog` were coded and documented in the same style as the other megawidgets. Each widget includes a test, a demonstration, the documentation source file, the documentation html file, and the code itself. Included here are only the documentation, the demonstration and the code. Use the *html* version of this document to browse the Pmw folders.

Pmw documentation on the web:

- Home page: `http://pmw.sourceforge.net/`
- How to Build widgets: `http://pmw.sourceforge.net/doc/howtobuild.html`

11.2 MultiListBox

Name

Pmw.MultiListBox() - multiple connected listboxes with interactive resizing.

Inherits

Pmw.MegaWidget

Description

A multilistbox consists of a *Pmw.PanedWidget*, a scrollbar, and an optional frame for labels. Each pane is packed with a standard listbox. A label can be attached to each listbox and placed in the labelframe above the paned widget. All the listboxes are connected, so that a selection in one listbox is a selection in all the listboxes, and if one listbox is scrolled, all the other listboxes will be scrolled. (This applies only in the y-direction, the listboxes scroll individually in the x-direction). The panes can be interactively resized.

Options

Options for this megawidget and its base classes are described below.

dblclickcommand This specifies a function to call when mouse button 1 is double clicked over an entry in one of the listbox components. The default is `None`.

labelheight Initialisation option. If the nolabels option is `False`, this specifies the height of the labelframe component. The default is 20.

labelmargin Initialisation option. If the nolabels option is `False`, this specifies the distance between the labelframe component and the pane component. The default is 2.

nolabels Initialisation option. If set to `True`, no labels will be drawn. The default is 0.

scrollmargin Initialisation option. The distance between the scrollbar component and the pane component. The default is 2.

selectioncommand This specifies a function to call when mouse button 1 is single clicked over an entry in one of the listbox components or if the <Space> or <Return> key is hit while one of the listboxes has focus. The default is `None`.

selectmode This is the same as the selectmode option for the *Tkinter.Listbox* widget, but this applies to all the listboxes together. This value is forwarded to the listbox components when they are created with the `addlistbox()` and `insertlistbox()` methods. Consult documentation on *Tk/Tkinter* to get a description of this option. The default is 'browse'.

separatorcolor The color of the lines separating the panes. The default is 'black'.

separatorthickness Initialisation option. Specifies the thickness of the lines separating the panes. The default is 1.

usehullsize Initialisation option. If true, the size of the megawidget is determined solely by the width and height options of the hull component.

Otherwise, the size of the megawidget is determined by the width and height of the pane component, along with the size and/or existence of the other components, such as the labelframe, the scrollbar and the scrollmargin option. All these affect the overall size of the megawidget. The default is 0.

Components

Components created by this megawidget and its base classes are described below.

hull This acts as the body for the entire megawidget. Other components are created as children of the hull to further specialise this class. By default, this component is a *Tkinter.Frame*.

labelframe If the nolabels option is `False` this component acts as a container for *Tkinter.Labels*. The labels will be dynamically created with the `addlistbox()` and `insertlistbox()` methods. The positions of the labels will be determined by the

corresponding positions (and sizes) of the panes. By default, this component is a *Tkinter.Frame*.

pane This component manages the resizable frames (panes). Each pane acts as a container for a *Tkinter.Listbox*, that will be dynamically created with the `addlistbox()` and `insertlistbox()` methods. The panes are arranged horizontally. By default, this component is a *Pmw.PanedWidget*.

scrollbar The vertical scrollbar, used to scroll all the listboxes. [At present time, there is no scrollmode (dynamic, static, none) available. The scrollbar will always be drawn.] By default, this component is a *Tkinter.Scrollbar*.

Dynamic components

Label components are created dynamically by the `addlistbox()` and `insertlistbox()` methods. By default, these are of type *Tkinter.Label* and are created with a component group of Label.

Listbox components are created dynamically by the `addlistbox()` and `insertlistbox()` methods. By default, these are of type *Tkinter.Listbox* and are created with a component group of Listbox.

Methods

Only methods specific to this megawidget are described below. For a description of its inherited methods, see the manual for its base class *Pmw.MegaWidget*. In addition, methods from the *Pmw.PanedWidget* class are forwarded by this megawidget to the pane component.

`activate(index)` Activate the given index (it will be marked with an underline). The active item can be refered to using the "active" index. This is a specialisation of the `Tkinter.Listbox.activate()` method.

`add(*args, **kw)` Same as `addlistbox()` method. This will override the paned widget's method with the same name.

`addlistbox(name, **kw)` Calling this method is equivalent to calling `insertlistbox()` with `before` set to the current number of panes/listboxes.

`clear()` Delete all items in the megawidget. Equivalent to `setlist(())` or `delete(0, 'end')`.

`configureitem(row, column, item)` Replace the item in (row, column) with the given item.

`configurelabel(name, **kw)` Configure the label specified by name, where name is either an integer, specifying the index of the label, or a string, specifying the name of the label. The keyword arguments specify the new values for the options for the label. If no keyword arguments are given, a dictionary containing the current settings for all label options will be returned.

`configurelistbox(name, **kw)` Configure the listbox specified by name, where name is either an integer, specifying the index of the listbox, or a string, specifying the name

of the listbox. The keyword arguments specify the new values for the options for the listbox. If no keyword arguments are given, a dictionary containing the current settings for all listbox options will be returned.

`configurerow(row, list)` Replace the items in the given row with the items in list.

`curselection()` Get a list of the currently selected alternatives. The list contains the indexes of the selected alternatives. This is a specialisation of the `Tkinter.Listbox.curselection()` method.

`delete(first, last = None)` Delete one or more rows. This will override the paned widget's method with the same name. This is a specialisation of the `Tkinter.Listbox.delete()` method.

`deletelistbox(name)` Delete the listbox specified by name, where name is either an integer, specifying the index of the listbox, or a string, specifying the name of the listbox. This will remove the corresponding pane and label.

`get(first = None, last = None)` This method will return a tuple of items, where the items are the row values specified by first and last. If first is `None` all rows are returned.

`getcurselection()` Same as `getvalue()` method.

`getvalue()` Return a tuple of tuples, where each tuple consists of the currently selected items of each listbox.

`index(index)` Return the numerical index corresponding to the given index. This is a specialisation of the `Tkinter.Listbox.index()` method.

`insert(index, list)` Insert the items in list at the given index. The items will be inserted into the listboxes from left to right. The number of items in list must be the same as the number of listboxes created. This will override the paned widget's method with the same name. This is a specialisation of the `Tkinter.Listbox.insert()` method.

`insertlistbox(name, before = 0, **kw)` Add a listbox as a component named name. The listbox is added just before the listbox specified by before, where before may be either an integer, specifying the index of the listbox, or a string, specifying the name of the listbox. A pane with component name "pane-" + name will be added to the paned widget component, and the new listbox will be placed inside this pane. If the nolabels option is `False`, a label with component name "label-" + name will also be created.

The keyword arguments are split into three. The keywords starting with "listbox_" are sent to the listbox component, the keywords starting with "label_" are sent to the label component, while the others are sent to the paned widget component's `add()` method.

To add a listbox to the end of the paned widget, use `add()` or `addlistbox()`.

`label(name)` Return the *Tkinter.Label* widget for the label specified by name, where name is either an integer, specifying the index of the label, or a string specifying the name of the label.

`labels()` Return a list of the names of the labels, in display order.

`listbox(name)` Return the *Tkinter.Listbox* widget for the listbox specified by name, where name is either an integer, specifying the index of the listbox, or a string specifying the name of the listbox.

`listboxes()` Return a list of the names of the listboxes, in display order.

`move(*args, **kw)` Same as `movelistbox()` method. This will override the paned widget's method with the same name.

`movelistbox(name, newPos, newPosOffset = 0)` Move the listbox specified by `name` to the new position specified by `newPos`. The first two arguments may be either an integer, specifying the index of the listbox, or a string, specifying the name of the listbox. If `newPosOffset` is specified, it is added to the `newPos` index.

`nearest(y)` Return the index nearest to the given coordinate (a widget-relative pixel coordinate). This is a specialisation of the `Tkinter.Listbox.nearest()` method.

`scan_dragto(x, y)` Scroll the listbox widgets contents according to the given mouse coordinate. The text is moved 10 times the distance between the scanning anchor and the new position. This is a specialisation of the `Tkinter.Listbox.scan_dragto()` method.

`scan_mark(x, y)` Set the scanning anchor for fast horizontal scrolling to the given mouse coordinate. This is a specialisation of the `Tkinter.Listbox.scan_mark()` method.

`see(index)` Make sure the given index is visible. This is a specialisation of the `Tkinter.Listbox.see()` method.

`select_anchor(index)` Set the selection anchor to the given index. The anchor can be referred to using the "anchor" index. This is a specialisation of the `Tkinter.Listbox.-select_anchor()` method.

`select_clear(first, last = None)` Clear the selection. This is a specialisation of the `Tkinter.Listbox.select_clear()` method.

`select_includes(index)` Returns true if the row at the given index is selected. This is a specialisation of the `Tkinter.Listbox.select_includes()` method.

`select_set(first, last = None)` Add one or more rows to the selection. This is a specialisation of the `Tkinter.Listbox.select_set()` method.

`setlist(list)` Replace all the items in all the listboxes with those specified by `list`, where `list` must be a sequence of row items. The number of items in each row must be the same as the number of listboxes created.

`size()` Return a tuple where the first item is the number of listboxes in the megawidget and the second item is the number of items in the listboxes. This is a specialisation of the `Tkinter.Listbox.size()` method.

`yview(*args)` This is a specialisation of the `Tkinter.Listbox.yview()` method. Depending on the number of arguments, it has four different applications.

`yview()` Determine which part of the listboxes is visible in the vertical direction. This

is given as the offset and size of the visible part, given in relation to the full size of the listboxes (1.0 = full size). This method is used by the scrollbar component.

`yview(index)` Adjust the listboxes so that the given index is at the top edge of the listboxes. To make sure that a given index is visible, use the `see()` method instead.

`yview("moveto", offset)` Adjust the listboxes so that the given offset is at the top edge of the listboxes. Offset 0.0 is the beginning, 1.0 the end. This method is used by the scrollbar component.

`yview("scroll", step, what)` Scroll the listboxes vertically by the given amount. The what argument can be either "units" or "pages". This method is used by the scrollbar component.

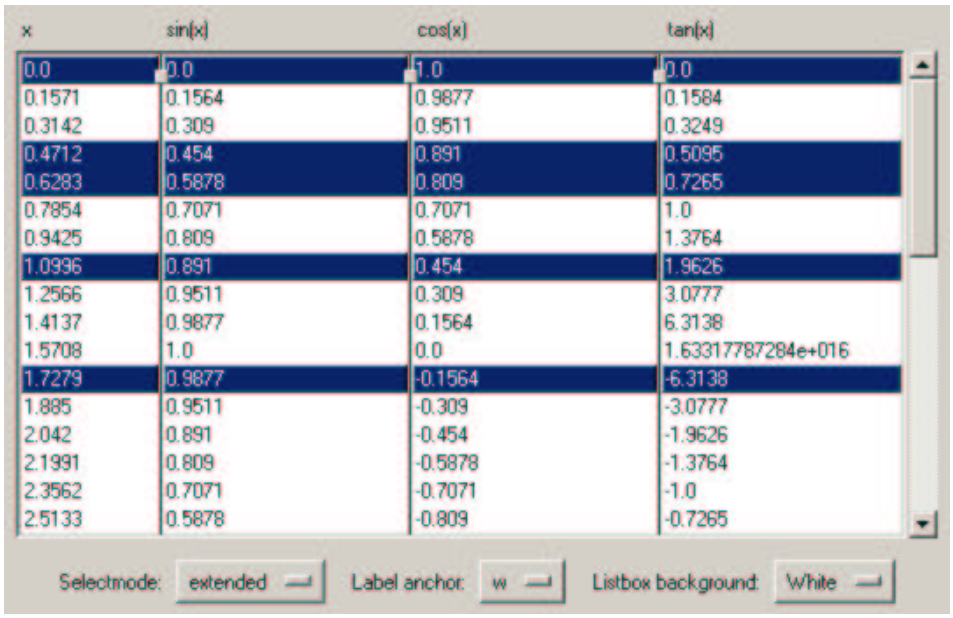


Figure 11.1: Snapshot created by the demonstration code.

Example

The figure 11.1 is a snapshot of the window (or part of the window) produced by the following code.

```
class Demo:
    def __init__(self, parent):
        # Create the MultiListbox
        self.mlb = Pmw.MultiListBox(parent,
                dblclickcommand = self.dblclick,
                hull_width = 525,
                hull_height = 300,
                selectioncommand = self.selection,
                usehullsize = 1)

        # Add some listboxes
        self.mlb.addlistbox('x', size = .16, min = .1)
```

```
        self.mlb.addlistbox('sin(x)', size = .28)
        self.mlb.addlistbox('cos(x)', size = .28)
        self.mlb.addlistbox('tan(x)', size = .28)

        self.mlb.pack(padx = 5, pady = 5, expand = 1, fill='both')

        # Fill the multilistbox
        for i in range(0, 41):
            x=i*2*math.pi/40
            s=str(round(math.sin(x), 4))
            c=str(round(math.cos(x), 4))
            t=str(round(math.tan(x), 4))
            self.mlb.insert('end', (str(round(x, 4)), s, c, t))

        # Create a frame for the option menus
        frame = Tkinter.Frame(parent)
        frame.pack(padx = 5, pady = 5)
        # Add option menu: selectmode
        self.selectmodeMenu = Pmw.OptionMenu(frame,
                command = self.change_selectmode,
                items = ('single', 'browse', 'multiple', 'extended'),
                labelpos = 'w',
                label_text = 'Selectmode: ')
        self.selectmodeMenu.pack(side = 'left', padx = 5)
        self.selectmodeMenu.invoke('extended')

        # Add option menu: label anchor
        self.labelAnchorMenu = Pmw.OptionMenu(frame,
                command = self.change_label_anchor,
                items = ('center', 'n', 'ne', 'e', 'se', 's', 'sw', 'w' ,'nw'),
                labelpos = 'w',
                label_text = 'Label anchor: ')
        self.labelAnchorMenu.pack(side = 'left', padx = 5)
        self.labelAnchorMenu.invoke('w')

        # Add option menu: listbox background
        self.listboxBgMenu = Pmw.OptionMenu(frame,
                command = self.change_listbox_bg,
                items = ('White', 'Gray', 'Red', 'Blue', 'Purple', 'Yellow'),
                labelpos = 'w',
                label_text = 'Listbox background: ')
        self.listboxBgMenu.pack(side = 'left', padx = 5)
        self.listboxBgMenu.invoke('White')

    def change_selectmode(self, mode):
        self.mlb.configure(selectmode = mode)

    def change_label_anchor(self, anchor):
        for label in self.mlb.labels():
            self.mlb.configurelabel(label, anchor = anchor)

    def change_listbox_bg(self, bg):
        for listbox in self.mlb.listboxes():
           self.mlb.configurelistbox(listbox, bg = bg)

    def selection(self):
        print 'Current selection: ',
        print self.mlb.curselection()
```

```
    def dblclick(self):
        print 'Items selected: ',
        print self.mlb.getcurselection()
```

Code

```
# MultiListBox megawidget written by Morten Fagerland

import types
import Tkinter
import Pmw

class MultiListBox(Pmw.MegaWidget):
    """ A ScrolledListBox megawidget with multiple scalable columns
    """

    def __init__(self, parent = None, **kw):

        # Define the megawidget options
        INITOPT = Pmw.INITOPT
        optiondefs = (
            ('dblclickcommand',     None,           None),
            ('labelmargin',         2,              INITOPT),
            ('labelheight',         20,             INITOPT),
            ('nolabels',            0,              INITOPT),
            ('scrollmargin',        2,              INITOPT),
            ('selectioncommand',    None,           None),
            ('selectmode',          'browse',       self._changeSelectmode),
            ('separatorcolor',      'black',        self._configSepColor),
            ('separatorthickness',  1,              INITOPT),
            ('usehullsize',         0,              INITOPT),
        )
        self.defineoptions(kw, optiondefs,
                dynamicGroups = ('Listbox', 'Label'))

        # Initialise base class (after defining options)
        Pmw.MegaWidget.__init__(self, parent)

        # Create the components
        interior = self.interior()

        if self['usehullsize']:
            interior.grid_propagate(0)

        if self['nolabels']:
            height = 0
        else:
            height = self['labelheight']

        self._labelframe = self.createcomponent('labelframe', (), None,
                Tkinter.Frame, interior,
                        height = height)

        self._pane = self.createcomponent('pane', (), None,
                Pmw.PanedWidget, interior,
                        command = self._changePaneSizes,
                        hull_borderwidth = 2,
                        hull_relief = 'sunken',
                        orient = 'horizontal',
```

```
                        separatorthickness = self['separatorthickness'],
                        )

        self._scrollbar = self.createcomponent('scrollbar', (), None,
                Tkinter.Scrollbar, interior,
                        orient = 'vertical',
                        command = self.yview)

        # Position all components
        self._labelframe.grid(row = 0, column = 0, sticky = 'ew')
        if not self['nolabels']:
            interior.grid_rowconfigure(1, minsize = self['labelmargin'])
        else:
            interior.grid_rowconfigure(1, minsize = 0)
        self._pane.grid(row = 2, column = 0, sticky = 'news')
        interior.grid_columnconfigure(1, minsize = self['scrollmargin'])
        self._scrollbar.grid(row = 2, column = 2, sticky = 'news')

        # Enable the labelframe to expand in the x-direction,
        # the scrollbar to expand in the y-direction,
        # and the pane to expand in both directions
        interior.grid_rowconfigure(2, weight = 1)
        interior.grid_columnconfigure(0, weight = 1)

        # List of tuples describing the listboxes and the labels
        # (name, widget)
        self._listboxList = []
        self._labelList = []

        # To avoid the selectioncommand being called both before and
        # after the dblclickcommand.
        self.lastEventWasDouble = 0

        # Check keywords and initialise options
        self.initialiseoptions()


#---------------
# Public methods
#---------------

    # I have included this only because it is a conveniant method name
    def add(self, *args, **kw):
        return apply(self.addlistbox, args, kw)

    def addlistbox(self, name, **kw):
        return apply(self.insertlistbox,
                     (name, len(self._listboxList)), kw)

    def clear(self):
        self.delete(0, 'end')

    def configureitem(self, row, column, item):
        listbox = self._listboxList[column][1]
        listbox.delete(row)
        listbox.insert(row, item)

    def configurelabel(self, name, **kw):
        index = self._nameToIndex(name, list = 'label')
```

```
        if not kw:
            return self._labelList[index][1].configure()
        else:
            return self._labelList[index][1].configure(kw)

    def configurelistbox(self, name, **kw):
        index = self._nameToIndex(name)
        if not kw:
            return self._listboxList[index][1].configure()
        else:
            return self._listboxList[index][1].configure(kw)

    def configurerow(self, row, list):
        self.delete(row)
        self.insert(row, list)

    def deletelistbox(self, name):
        name = self._indexToName(name)
        index = self._nameToIndex(name)

        # Remove listbox and label components
        self.destroycomponent(name)
        self.destroycomponent('label-'+name)

        # Update listbox and label list info
        del self._listboxList[index]
        del self._labelList[index]

        # Remove pane and update/redraw
        self._pane.delete(index)
        self._pane.updatelayout()

    def get(self, first = None, last = None):
        list = []
        for name,listbox in self._listboxList:
            if first == None:
                list.append(listbox.get(0, 'end'))
            else:
                list.append(listbox.get(first, last))
        if (first != None and last == None):
            return tuple(list)

        # Return a list of the rows (not a list of the columns)
        newlist = []
        for i in range(len(list[0])):
            newlist.append([])
            for n in range(len(list)):
                newlist[i].append(list[n][i])
        return tuple(map(tuple, newlist))

    def getcurselection(self):
        if not len(self._listboxList):
            return ()
        selection = self._listboxList[0][1].curselection()
        list = []
        for i in range(len(selection)):
            list.append([])
            for name,listbox in self._listboxList:
                list[i].append(listbox.get(selection[i]))
```

```
        return tuple(map(tuple, list))

    def getvalue(self):
        return self.getcurselection()

    def insertlistbox(self, name, before = 0, **kw):
        if name in self.components():
            raise ValueError, 'listbox "%s" already exists' % name

        # Remove label keywords (and set some default ones)
        labelkw = {}
        labelkw['anchor'] = 'w'
        labelkw['text'] = name
        for key,value in kw.items():
            if key.startswith('label_'):
                labelkw[key[6:]] = value
                del kw[key]

        # Remove listbox keywords (and set some default ones)
        listboxkw = {}
        listboxkw['borderwidth'] = 0
        listboxkw['relief'] = 'flat'
        for key,value in kw.items():
            if key.startswith('listbox_'):
                listboxkw[key[8:]] = value
                del kw[key]
        # Override these keywords (or face the consequences!)
        listboxkw['exportselection'] = 0
        listboxkw['selectmode'] = self['selectmode']

        # Forward the rest of the keywords to the PanedWidget's insert method
        index = self._nameToIndex(before)
        self._pane.insert('pane-'+name, before = index, **kw)

        # Configure the pane separators' color
        self._configSepColor()

        # Create the label component
        label = self.createcomponent('label-'+name, (), 'Label',
                Tkinter.Label, self._labelframe, labelkw)
        label.place(height = self['labelheight'])
        self._labelList.insert(index, ('label-'+name, label))

        # This will update the label positions
        self._pane.updatelayout()

        # Create the listbox component
        listbox = self.createcomponent(name, (), 'Listbox',
                Tkinter.Listbox, self._pane.pane('pane-'+name), listboxkw)
        listbox.grid(row = 0, column = index, sticky = 'news')
        self._listboxList.insert(index, (name, listbox))

        # Handle listbox scrolling
        listbox.configure(yscrollcommand = Command(self._scrollListboxes,
                listbox))

        # Enable the listbox to expand in both directions
        self._pane.pane('pane-'+name).grid_rowconfigure(0, weight = 1)
        self._pane.pane('pane-'+name).grid_columnconfigure(index, weight = 1)
```

```
        # Define LMB callbacks
        listbox.bind('<Button-1>', Command(self._select, listbox))
        listbox.bind('<Shift-Button-1>', Command(self._selectShift, listbox))
        listbox.bind('<Control-Button-1>',
                     Command(self._selectControl, listbox))
        listbox.bind('<Button1-Motion>', Command(self._selectMotion, listbox))
        listbox.bind('<Control-Button1-Motion>',
                     Command(self._selectControlMotion, listbox))
        listbox.bind('<ButtonRelease-1>', self._release)
        listbox.bind('<Double-1>', self._dblclick, '+')

        # Bind space and return keys
        listbox.bind('<Key-space>', self._release)
        listbox.bind('<Key-Return>', self._release)

    def label(self, name):
        index = self._nameToIndex(name, list = 'label')
        return self._labelList[index][1]

    def labels(self):
        labels = []
        for name,label in self._labelList:
            labels.append(name)
        return labels

    def listbox(self, name):
        index = self._nameToIndex(name)
        return self._listboxList[index][1]

    def listboxes(self):
        listboxes = []
        for name,listbox in self._listboxList:
            listboxes.append(name)
        return listboxes

    # I have included this only because it is a conveniant method name
    def move(self, *args, **kw):
        return apply(self.movelistbox, args, kw)


    def movelistbox(self, name, newPos, newPosOffset = 0):
        oldPos = self._nameToIndex(name)
        newPos = self._nameToIndex(newPos) + newPosOffset
        if newPos < 0 or newPos >=len(self._pane.panes()):
            return

        # Update listbox list info
        name,listbox = self._listboxList[oldPos]
        del self._listboxList[oldPos]
        self._listboxList.insert(newPos, (name, listbox))

        # Update label list info
        name,label = self._labelList[oldPos]
        del self._labelList[oldPos]
        self._labelList.insert(newPos, (name, label))

        # Move the pane to the new postition
        # (this will update the label postitions too)
```

```
        self._pane.move(oldPos, newPos)

    def setlist(self, list):
        self.delete(0, 'end')
        for row in list:
            self.insert('end', row)


# --------------------------------------------------------------------
# Methods from the Tkinter.Listbox that make sense for all the listboxes
# --------------------------------------------------------------------

    def activate(self, index):
        for name,listbox in self._listboxList:
            listbox.activate(index)

    def curselection(self):
        if not self._listboxList:
            return ()
        # The selection should be the same in all the listboxes
        # => use the first listbox's curselection
        return self._listboxList[0][1].curselection()

    # Do not confuse with the deletelistbox() method!
    # This method deletes listbox items, not listboxes
    def delete(self, first, last = None):
        for name,listbox in self._listboxList:
            listbox.delete(first, last)

    def index(self, index):
        if not self._listboxList:
            raise ValueError, 'no listboxes have been created'
        # This value should be the same in all the listboxes
        # => use the first listbox's index
        return self._listboxList[0][1].index(index)

    # Do not confuse with the insertlistbox() method!
    # This method inserts listbox items, not listboxes
    def insert(self, index, list):
        if not self._listboxList:
            raise ValueError, 'no listboxes have been created'
        for i in range(len(self._listboxList)):
            self._listboxList[i][1].insert(index, str(list[i]))

        # Use this instead if you want to add a little
        # space to the right of the pane separator.
        # You should probably do the same with the labels too
#        self._listboxList[0][1].insert(index, str(list[0]))
#        for i in range(1, len(self._listboxList)):
#            self._listboxList[i][1].insert(index, '  '+str(list[i]))

    def nearest(self, y):
        if not self._listboxList:
            raise ValueError, 'no listboxes have been created'
        # This value should be the same in all the listboxes
        # => use the first listbox's index
        return self._listboxList[0][1].nearest(y)

    def see(self, index):
```

```
        if not self._listboxList:
            raise ValueError, 'no listboxes have been created'
        for name,listbox in self._listboxList:
            listbox.see(index)

    def size(self):
        if not len(self._listboxList):
            return (0,0)
        return (len(self._listboxList), self._listboxList[0][1].size())


# Selection methods

    def select_anchor(self, index):
        for name,listbox in self._listboxList:
            listbox.select_anchor(index)

    def select_clear(self, first, last = None):
        for name,listbox in self._listboxList:
            listbox.select_clear(first, last)

    def select_includes(self, index):
        if not self._listboxList:
            raise ValueError, 'no listboxes have been created'
        # This value should be the same in all the listboxes
        # => use the first listbox's value
        return self._listboxList[0][1].select_includes(index)

    def select_set(self, first, last = None):
        for name,listbox in self._listboxList:
            listbox.select_set(first, last)


# Scrolling methods

    def scan_mark(self, x, y):
        for name,listbox in self._listboxList:
            listbox.scan_mark(x, y)

    def scan_dragto(self, x, y):
```

11.3 ProgressBarDialog

Name

Pmw.ProgressBarDialog() - a dialog with a progress bar.

Inherits

Pmw.Dialog

Description

A dialog with a progress bar and optional label and buttons. The progress bar may be updated with the `updatebarlength()` method.

Options

Options for this megawidget and its base classes are described below.

activatecommand If this is callable, it will be called whenever the megawidget is activated by a call to `activate()`. The default is `None`.

barfill The colour to use for the interior of the bar. The default is "blue2".

baroutline The colour to use for the outline of the bar. The default is "blue2".

baroutlinewidth The width of the outline. The default is 1.

barstipple The name of a bitmap which is used as a stipple brush when filling the interior of the bar. The default is `None`.

borderx Initialisation option. The padding to the left and right of the bar. The default is 10.

bordery Initialisation option. The padding above and below the bar. The default is 10.

buttonboxpos Initialisation option. Specifies on which side of the dialog window to place the button box. Must be one of "n", "s", "e" or "w". The default is "s".

buttons This must be a tuple or a list and specifies the names on the buttons in the button box. The default is `('OK',)`.

command Specifies a function to call whenever a button in the button box is invoked or the window is deleted by the window manager. The function is called with a single argument, which is the name of the button which was invoked, or `None` if the window was deleted by the window manager.

If the value of command is not callable, the default behaviour is to deactivate the window if it is active, or withdraw the window if it is not active. If it is deactivated, `deactivate()` is called with the button name or `None` as described above. The default is `None`.

deactivatecommand If this is callable, it will be called whenever the megawidget is deactivated by a call to `deactivate()`. The default is `None`.

defaultbutton Specifies the default button in the button box. If the <Return> key is hit when the dialog has focus, the default button will be invoked. If defaultbutton is `None`, there will be no default button and hitting the <Return> key will have no effect. The default is `None`.

labelmargin Initialisation option. If the labelpos option is not `None`, this specifies the distance between the label component and the rest of the megawidget. The default is 0.

labelpos Initialisation option. Specifies where to place the label component. If not `None`, it should be a concatenation of one or two of the letters "n", "s", "e" and "w". The first letter specifies on which side of the megawidget to place the label. If a second letter is specified, it indicates where on that side to place the label. For example, if labelpos is "w", the label is placed in the center of the left hand side; if it is "wn", the

label is placed at the top of the left hand side; if it is "ws", the label is placed at the bottom of the left hand side.

If `None`, a label component is not created. The default is `None`.

master This is used by the `activate()` method to control whether the window is made transient during modal dialogs. See the `activate()` method. The default is "parent".

separatorwidth Initialisation option. If this is greater than 0, a separator line with the specified width will be created between the button box and the child site, as a component named separator. Since the default border of the button box and child site is raised, this option does not usually need to be set for there to be a visual separation between the button box and child site. The default is 0.

textfill The colour to use for the text specifying the progress of the bar. This should be a list of two elements, specifying the color to use when the progress (see the `updatebarlength()` method) is less than or equal to .5, and greater than .5 respectively. The default is `['black', 'white']`.

title This is the title that the window manager displays in the title bar of the window. The default is `None`.

Components

Components created by this megawidget and its base classes are described below.

bar The rectangle in the middle of the megawidget displaying the progress. By default it is created with the options `(borderwidth = 2, height = 18, relief = 'sunken', width = 200)`. By default, this component is a *Tkinter.Canvas*.

buttonbox This is the button box containing the buttons for the dialog. By default it is created with the options `(hull_borderwidth = 1, hull_relief = 'raised')`. By default, this component is a *Pmw.ButtonBox*.

dialogchildsite This is the child site for the dialog, which may be used to specialise the megawidget by creating other widgets within it. By default it is created with the options `(borderwidth = 1, relief = 'raised')`. By default, this component is a *Tkinter.Frame*.

hull This acts as the body for the entire megawidget. Other components are created as children of the hull to further specialise this class. By default, this component is a *Tkinter.Toplevel*.

label If the labelpos option is not `None`, this component is created as a text label for the megawidget. See the labelpos option for details. Note that to set, for example, the text option of the label, you need to use the label_text component option. By default, this component is a *Tkinter.Label*.

Methods

Only methods specific to this megawidget are described below. For a description of its inherited methods, see the manual for its base class *Pmw.Dialog*.

`updatebarlength(progress)` Set the length of the progress bar according to progress, a number between 0.0 and 1.0, specifying a part of the total length of the bar.

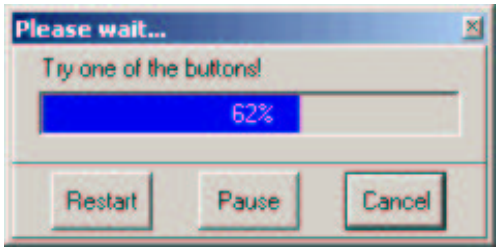


Figure 11.2: Snapshot created by the demonstration code.

Example

The figure 11.2 is a snapshot of the window (or part of the window) produced by the following code.

```
class Demo:
    def __init__(self, parent):
        self.parent = parent
        self.pause = Tkinter.IntVar()

        # Create buttons
        buttonbox = Pmw.ButtonBox(parent, orient = 'vertical')
        buttonbox.pack(padx = 10, pady = 5)
        buttonbox.add('Simple dialog', command = self.simpleDialog)
        buttonbox.add('Dialog with label', command = self.labelDialog)
        buttonbox.add('Dialog with button', command = self.buttonDialog)
        buttonbox.add('Dialog with buttons', command = self.buttonsDialog)
        buttonbox.add('Dialog with label and buttons',
                command = self.labelButtonsDialog)

        # Create scale
        self.scale = Tkinter.Scale(parent, label = 'Size (in k) of test loop',
                from_ = 1,
                length = 240,
                orient = 'horizontal',
                resolution = .5,
                tickinterval = 1.5,
                to = 10)
        self.scale.pack(padx = 10, pady = 5)
        self.scale.set('5.5')

    def simpleDialog(self):
        kw = {'buttons': ()}
        self.createDialog(**kw)

    def labelDialog(self):
        kw = {'buttons': (),
              'labelpos': 'n',
              'label_text': 'Put some text here'}
        self.createDialog(**kw)
```

```
    def buttonDialog(self):
        kw = {'buttons': ('Cancel',),
              'defaultbutton': 0}
        self.createDialog(**kw)

    def buttonsDialog(self):
        kw = {'buttons': ('Restart', 'Pause', 'Cancel'),
              'defaultbutton': 2}
        self.createDialog(**kw)

    def labelButtonsDialog(self):
        kw = {'buttons': ('Restart', 'Pause', 'Cancel'),
              'defaultbutton': 2,
              'labelpos': 'nw',
              'label_text': 'Try one of the buttons!'}
        self.createDialog(**kw)

    # Create the progress bar dialog on the fly and start a loop
    def createDialog(self, **kw):
        kw['command'] = self.buttonPress
        kw['title'] = 'Please wait...'

        self.dialog = Pmw.ProgressBarDialog(self.parent, **kw)
        self.dialog.transient(self.parent)
        self.dialog.focus()

        self.max = self.scale.get() * 1000
        self.pause.set(0)
        self.index = 0
        while self.index < self.max:
            if self.pause.get():
                self.parent.wait_variable(self.pause)
            progress = float(self.index)/(self.max-1)
            self.dialog.updatebarlength(progress)
            self.index = self.index + 1
        self.dialog.destroy()

    def buttonPress(self, button):
        if button == 'Restart':
            self.index = 0
            self.pause.set(0)
        elif button == 'Pause':
            self.pause.set(not self.pause.get())
        else:
            self.index = self.max
            self.dialog.destroy()
```

Code

```
# ProgressBarDialog megawidget written by Morten Fagerland

import Tkinter
import Pmw

class ProgressBarDialog(Pmw.Dialog):
    """ A dialog with a progress bar
    """

    def __init__(self, parent = None, **kw):
```

```
        # Define the megawidget options
        INITOPT = Pmw.INITOPT
        CONFIG = self._configBar
        optiondefs = (
            ('barfill',             'blue2',               CONFIG),
            ('baroutline',          'blue2',               CONFIG),
            ('baroutlinewidth',     1,                     CONFIG),
            ('barstipple',          None,                  CONFIG),
            ('borderx',             10,                    INITOPT),
            ('bordery',             10,                    INITOPT),
            ('labelmargin',         0,                     INITOPT),
            ('labelpos',            None,                  INITOPT),
            ('textfill',            ['black', 'white'],    None),

            ('bar_borderwidth',     2,                     INITOPT),
            ('bar_height',          18,                    INITOPT),
            ('bar_relief',          'sunken',              INITOPT),
            ('bar_width',           200,                   INITOPT),
        )
        self.defineoptions(kw, optiondefs)

        # Initialise the base class (after defining the options)
        Pmw.Dialog.__init__(self, parent)

        # Create the components.
        interior = self.interior()

        self.createlabel(interior)

        self._bar = self.createcomponent('bar', (), None,
                Tkinter.Canvas, interior)

        # Create the actual bar as a canvas rectangle
        # and the progress text as a canvas text
        self._bar.create_rectangle(0,0, 0,0, tags = 'bar')
        self._bar.create_text(0,0, tags = 'text')

        # Resize the canvas rectangle if the bar is resized
        self._bar.bind('<Configure>', self._resizeBar)

        # Position components
        interior.grid_rowconfigure(0, minsize = self['bordery'])
        interior.grid_rowconfigure(4, minsize = self['bordery'])
        interior.grid_columnconfigure(0, minsize = self['borderx'])
        interior.grid_columnconfigure(4, minsize = self['borderx'])
        self._bar.grid(row = 2, column = 2, sticky = 'news')

        # Allow the progress bar to expand in both directions
        interior.grid_columnconfigure(2, weight = 1)
        interior.grid_rowconfigure(2, weight = 1)

        # Keep the last progress value (used when the canvas has been resized)
        self._lastProgressValue = 0

        # Check keywords and initialise options
        self.initialiseoptions()

    def updatebarlength(self, progress):
```

```
        self._lastProgressValue = progress
        width = self._bar.winfo_width() - 5
        height = self._bar.winfo_height() - 5

        # Update bar
        self._bar.coords('bar', 4,4, progress*width,height)

        # Update progress text (if halfway: change color)
        if progress <= .5:
            fill = self['textfill'][0]
        else:
            fill = self['textfill'][1]
        text = str(int(progress*100))+'%'
        self._bar.coords('text', 4 + width/2, 2 + height/2)
        self._bar.itemconfigure('text', text = text, fill = fill)

        self._bar.update()

    def _resizeBar(self, event):
        self.updatebarlength(self._lastProgressValue)

    def _configBar(self):
        self._bar.itemconfigure('bar',
                fill = self['barfill'],
                outline = self['baroutline'],
                stipple = self['barstipple'],
                width = self['baroutlinewidth'])
```

Part III

Applications

Chapter 12

Viscous flow between parallel plates (`fluidflow`)

12.1 The physical problem

Consider an incompressible fluid flowing between two parallel plates of infinite length (see figure 12.1). The plates are inclined an angle θ with respect to the x-axis and we assume a gravitational force $\mathbf{g}$ is exerted on the fluid. The distance between the plates is given by the value H and a constant pressure is applied in the direction of the x-axis. The pressure is described by the values β_L and β_R which are the sizes of the pressure gradient on the left and right hand side of the flow. The plates may move independently at a constant speed (U_0 and U_H) in the direction of the x-axis.

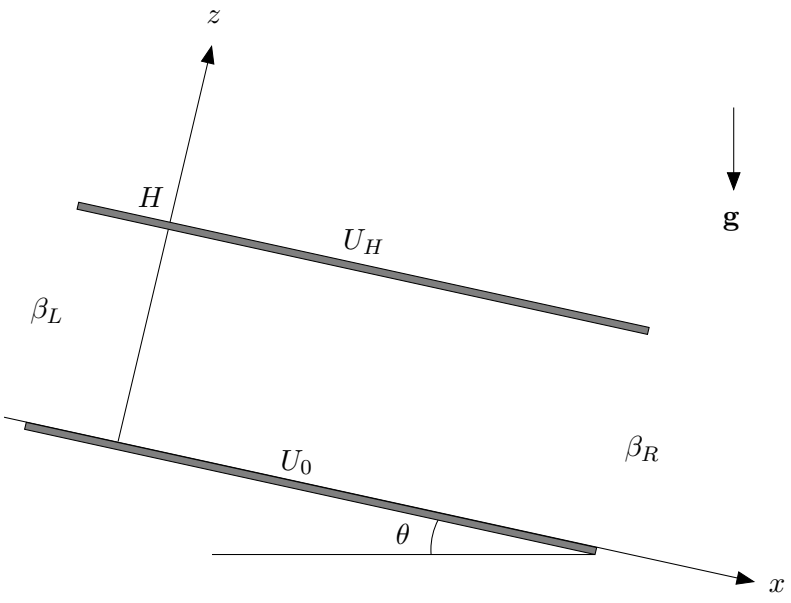


Figure 12.1: Flow between parallel plates.

12.2 The mathematical model

The movement of the fluid may be described by the Navier-Stokes equation (see for example [7] or [8])

$$\rho\left(\frac{\partial \mathbf{v}}{\partial t} + (\mathbf{v}\cdot\nabla)\mathbf{v}\right) = -\nabla p + \mu\nabla^2\mathbf{v} + \rho\mathbf{b}, \tag{2.1}$$

where ρ is the density, $\mathbf{v}$ the velocity of the fluid, ∇p the pressure gradient, μ the viscosity coefficient, and $\mathbf{b}$ the body forces working on the fluid. Let us assume that we have a stationary and straight path flow. This yields, in combination with the equation of continuity (incompressible fluid):

$$\mathbf{v} = u(z)\,\mathbf{i} \quad \Rightarrow \quad \mathbf{v}\cdot\nabla = 0,$$

which simplifies the Navier-Stokes equation:

$$\nabla^2\mathbf{v} = \frac{1}{\mu}\big(\nabla p - \rho\mathbf{b}\big). \tag{2.2}$$

The pressure is constant in the direction of the x-axis

$$\nabla p = \frac{\partial p}{\partial x}\,\mathbf{i}, \quad \frac{\partial p}{\partial x} = -\beta, \quad \beta = \beta_L - \beta_R,$$

and gravity is the the only body force present:

$$\mathbf{b} = \mathbf{g} = g\sin\theta\,\mathbf{i} - g\cos\theta\,\mathbf{k}.$$

The $\mathbf{i}$-component of the Navier-Stokes equations now becomes

$$u''(z) = -\frac{1}{\mu}\big(\beta + \rho g\sin\theta\big).$$

This equation may be integrated twice with respect to z:

$$u(z) = -\frac{1}{2\mu}\big(\beta + \rho g\sin\theta\big)z^2 + Az + B,$$

where A and B are constants to be determined. If we use the boundary conditions $u(0) = U_0$ and $u(H) = U_H$, A and B get the following values

$$\begin{aligned} A &= \frac{U_H - U_0}{H} + \frac{H}{2\mu}\big(\beta + \rho g\sin\theta\big), \\ B &= U_0. \end{aligned}$$

We can now state the explicit solution for the movement of the flow:

$$u(z) = -\frac{1}{2\mu}\big(\beta + \rho g\sin\theta\big)z^2 + \left(\frac{U_H - U_0}{H} + \frac{H}{2\mu}\big(\beta + \rho g\sin\theta\big)\right)z + U_0. \tag{2.3}$$

12.3 The illustration

The purpose of this application is to be able to change the value of the physical parameters and to see what kind of influence this will have on the flow profile. We have the following parameters:

- μ - viscosity coefficient
- ρ - fluid density
- β - size of pressure gradient
- θ - plate inclination
- H - distance between plates
- U_H - velocity of the upper plate
- U_0 - velocity of the lower plate
- g - gravitational constant

The application should easily visualise what type of flow we get with simple combinations of the parameters. Some suitable default values have been chosen and, to keep things simple, there is only possible to modify some basic properties of the flow to begin with: The plates may be set to move left, right or stand still. The inclination of the plates (θ) may be positive, negative or zero. The pressure may be greatest on the left side, on the right side or equal on both sides. In other words we may set the parameters β, θ, U_H and U_0 to be positive, negative or zero. This should be possible with clickable symbols so that some simple well-known flow profiles can be easily generated. In addition there should be possible to change the actual value of all the parameters in a separate window. In this way the influence of for example the viscosity may be studied for various types of flow.

12.4 The user interface

To visualise the fluid flow a number of evenly distributed arrows along the z-axis have been used. The number of these arrows can be changed with a *Pmw.Counter* widget. The flow profile may be visualised by how fast the arrows gain length; a velocity that is determined by the corresponding value $u(z)$. This will give an intuitive description of how the fluid behaves for different values of z, and the overall flow profile should be easily detected. The snapshots below (figures 12.2 - 12.5) feature some examples of different flow profiles. Animation of the flow may be started and stopped by clicking the button "[Start][Stop] fluid flow".

How to change the flow properties:

- There are three symbols at the top of the canvas that may be used to change the basic inclination of the plates. Just click on one of the symbols and the plates will tilt the appropriate way.

- Each plate has this set of symbols: "<<", "||", ">>" that may be used to control the plate's movement. The symbols corresponds to left movement, no movement, and right movement, respectively. Plate movement will be animated by a thick moving arrow running the length of the plate. This animation may be canceled by the "Animate plate movement?" checkbutton.
- The pressure may be set to either high or low on both sides of the flow by pressing the text labels. An equal value on both sides will make the pressure gradient zero.
- Press the "System properties" button to change the value of all the physical parameters mentioned above (section 12.3). A standard preference window (see the `dialogs` module, section 6) will appear.

Press the "Help" button to display the reference page (a *html* version of this section).

12.5 Some remarks about using threads

Using *threads* is a brilliant way to manage several blocks of code simultaneously. It is unfortunately not very compatible with *Tkinter*. It usually works, but it is not entirely stable and should be used with caution. The animation of the flow and the plate movement in this application are managed by threads. To avoid its use the plate movement animation could be canceled and the flow animation coded sequentially.

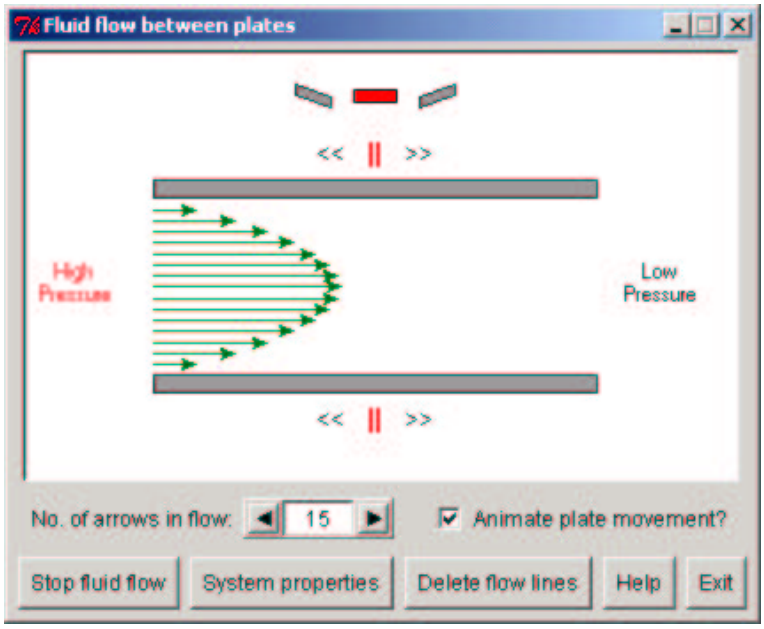


Figure 12.2: Pressure-driven flow.

12.6 Code

```
#!/bin/sh
""":"
```

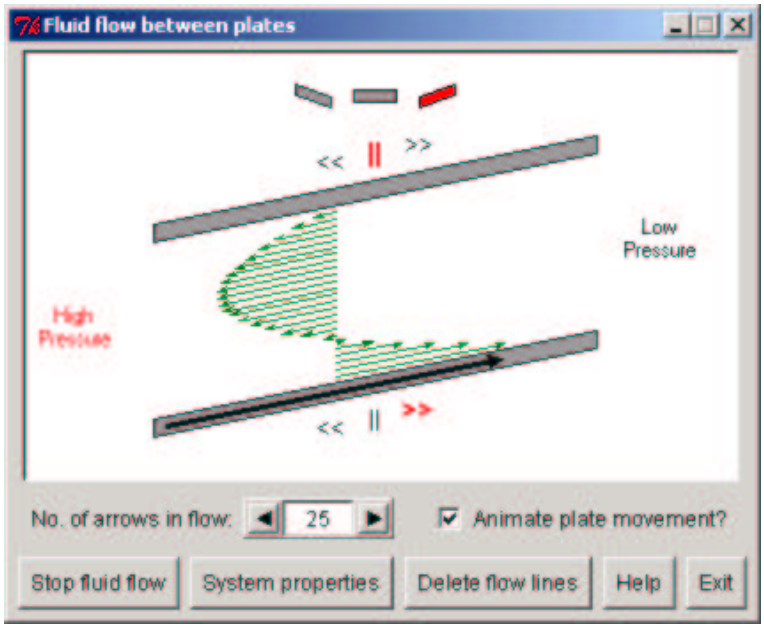


Figure 12.3: Flow driven by pressure, gravity and movement of the lower plate.

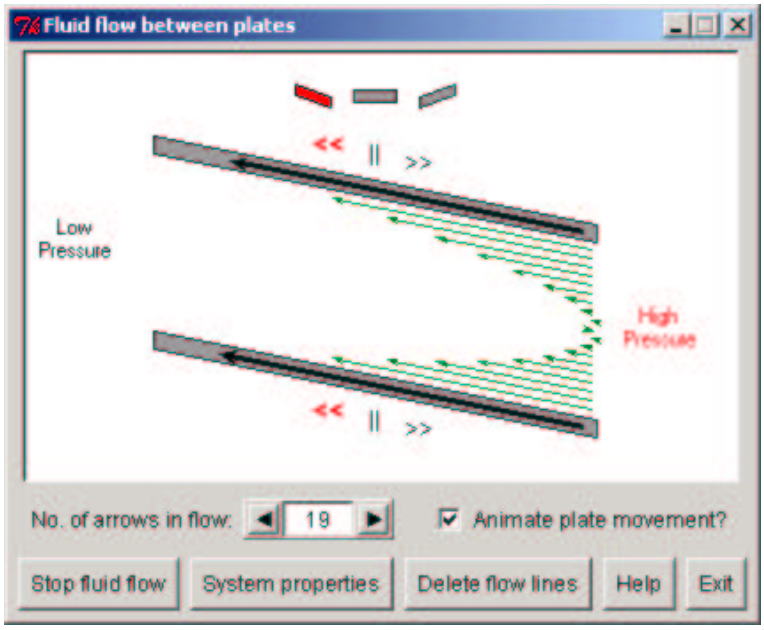


Figure 12.4: Flow driven by pressure, gravity and movement of both plates.

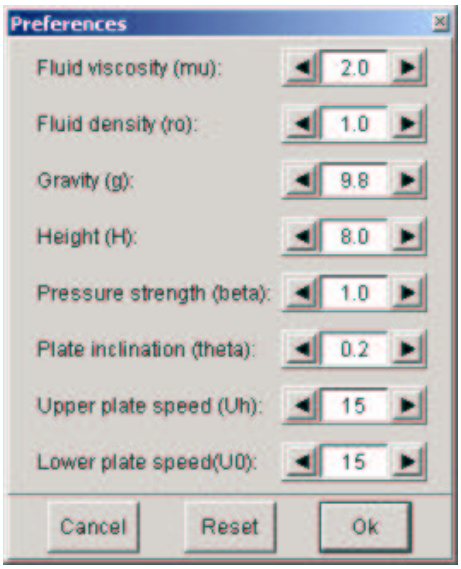


Figure 12.5: The preference window.

```
exec python $0 ${1+"$@"}
"""

# Make sure the modules are found
import sys
sys.path[:0] = ['../']

# This will enable the program to be run from idle
import os
dirname,basename = os.path.split(sys.argv[0])
os.chdir(os.path.normpath(dirname))

import Tkinter
import Pmw
import thread
import time
from modules import misc
from modules import dialogs
from math import *

class Gui:
    """Handle interface and user interaction"""

    def __init__(self):
        self.bindings()

        self.plateMovement = Tkinter.IntVar()
        self.plateMovement.set('1')    # Visualize plate movement?
        self.fluidFlow = 0             # Fluid flow animation in progress?
        self.tilt = 0                  # 1: left low, 0:horizontal, -1: left high
        self.top = 0                   # Top plate movement? (-1: left, 1:right)
        self.bottom = 0                # Bottom plate movement? (-1: left, 1:right)
        self.left = 0                  # Pressure at left side?
        self.right = 0                 # Pressure at right side?
        self.system = System()         # System parameters and solution
```

```
        # Create canvas
        self.canvas = Tkinter.Canvas(root,
                background = 'White',
                borderwidth = 2,
                height = 240,
                relief = 'sunken',
                width = 400)
        self.canvas.grid(row = 0, columnspan = 2, padx = 2, pady = 2)

        # Clean up if window is destroyed (all widgets are destroyed first)
        self.canvas.bind('<Destroy>', self.clean_up)

        # Create the number-of-arrows counter
        validator = {'validator': 'integer', 'min': '3', 'max': '30'}
        self.counter = Pmw.Counter(root,
                entryfield_validate = validator,
                entryfield_value = '13',
                entry_justify = 'center',
                entry_width = 5,
                labelpos = 'w',
                label_text = 'No. of arrows in flow: ')
        self.counter.grid(row = 1, column = 0, padx = 2, pady = 5)

        # Create the animate plate movement? checkbutton
        checkbutton = Tkinter.Checkbutton(root,
                command = self.check_plate_movement,
                text = 'Animate plate movement?',
                variable = self.plateMovement)
        checkbutton.grid(row = 1, column = 1, padx = 2)

        # Create the buttons
        if os.name == 'nt':
            padx = 3
        else:
            padx = 0
        buttons = Pmw.ButtonBox(root, padx = padx)
        self.startBtn = buttons.add('Start fluid flow',
                command = self.start_flow)
        buttons.add('System properties', command = self.properties)
        buttons.add('Delete flow lines', command = self.delete_flow_lines)
        buttons.add('Help', command = self.help)
        buttons.add('Exit', command = self.exit)
        buttons.grid(row = 2, column = 0, columnspan = 2, padx = 2, pady = 2)

        self.create_canvas_stuff()
        self.canvas.itemconfigure('top=0', font = 'Helvetica 11 bold')
        self.canvas.itemconfigure('bottom=0', font = 'Helvetica 11 bold')

    # Create keyboard bindings
    def bindings(self):
        root.bind('<q>', self.exit)
        root.bind('<h>', self.help)

    # Create plates, symbols and text inside the canvas
    def create_canvas_stuff(self):
        # Create plates
        coords = (75,75, 325,75, 325,85, 75,85, 75,75)
        kw = {'outline': 'Black', 'fill': 'Gray60', 'tags': 'plate_top'}
```

```
        self.canvas.create_polygon(coords, **kw)

        coords = (75,185, 325,185, 325,195, 75,195, 75,195)
        kw['tags'] = 'plate_bottom'
        self.canvas.create_polygon(coords, **kw)

        # Create tilt symbols
        list = (((155,21, 175,28, 175,34, 155,27, 155,21), 'Gray60','tilt=-1'),
                ((188,24, 212,24, 212,31, 188,31, 188,24), 'Red',   'tilt=0'),
                ((225,28, 245,21, 245,27, 225,34, 225,28), 'Gray60','tilt=1'))
        for coords,colour,tag in list:
            kw = {'outline': 'Black', 'fill': colour, 'tags': tag}
            self.canvas.create_polygon(coords, **kw)
            cmd = misc.Command(self.tilt_plates, tag)
            self.canvas.tag_bind(tag, '<Button-1>', cmd)

        # Create plate movement symbols
        list = ((175, 60,   'Black', ' << ', ('top=-1',      'top')),
                (200, 60,   'Red',   ' || ', ('top=0',       'top')),
                (225, 60,   'Black', ' >> ', ('top=1',       'top')),
                (175, 210,  'Black', ' << ', ('bottom=-1',   'bottom')),
                (200, 210,  'Red',   ' || ', ('bottom=0',    'bottom')),
                (225, 210,  'Black', ' >> ', ('bottom=1',    'bottom')))
        for x,y,color,text,tags in list:
            kw = {'fill': color, 'text': text, 'font': 'Helvetica 10 roman',
                  'tags': tags}
            self.canvas.create_text(x, y, **kw)
            cmd = misc.Command(self.plate_movement, tags)
            self.canvas.tag_bind(tags[0], '<Button-1>', cmd)

        # Create pressure texts
        pressure = ((10,120, 'left'), (340,120, 'right'))
        kw = {'anchor': 'nw', 'justify': 'center', 'text': 'Low\nPressure'}
        for x,y,tag in pressure:
            self.canvas.create_text(x, y, tags = tag, **kw)
            cmd = misc.Command(self.toggle_pressure, tag)
            self.canvas.tag_bind(tag, '<Button-1>', cmd)

    # Tilt plates, movement symbols and pressure texts
    def tilt_plates(self, event, tag):
        # If no change in tilt: return
        if self.tilt == int(tag[5:]):
            return

        # Change tilt symbol color and set variable
        for symbolTag in ('tilt=-1', 'tilt=0', 'tilt=1'):
            self.canvas.itemconfigure(symbolTag, fill = 'Gray60')
        self.canvas.itemconfigure(tag, fill = 'Red')
        exec 'self.'+tag

        # Stop fluid flow and plate movements
        self.fluidFlow = 0
        self.plate_movement(tags = ('top=0', 'top'))
        self.plate_movement(tags = ('bottom=0', 'bottom'))

        # Tilt walls
        t = self.tilt*25
        coords = (75,75+t, 325,75-t, 325,85-t, 75,85+t, 75,75+t)
        self.canvas.coords('plate_top', coords)
```

```
        coords = (75,185+t, 325,185-t, 325,195-t, 75,195+t, 75,185+t)
        self.canvas.coords('plate_bottom', coords)

        # Tilt pressure texts
        self.canvas.coords('left', 10, 120+t)
        self.canvas.coords('right', 340, 120-t)

        # Tile plate movement symbols
        self.canvas.coords('top=-1', 175, 60+t/5)
        self.canvas.coords('top=1', 225, 60-t/5)
        self.canvas.coords('bottom=-1', 175, 210+t/5)
        self.canvas.coords('bottom=1', 225, 210-t/5)

    # If the checkbutton is activated: animate plate movement (if necessary)
    def check_plate_movement(self):
        if not self.plateMovement.get():
            return
        if self.top:
            kw = {'tags': ('top='+str(self.top),'top'), 'skipTest': 1}
            self.plate_movement(**kw)
        if self.bottom:
            kw = {'tags': ('bottom='+str(self.bottom),'bottom'), 'skipTest': 1}
            self.plate_movement(**kw)

    # Start/stop plate movement animation
    def plate_movement(self, event = None, tags = None, skipTest = 0):
        plate,value = tags[0].split('=')
        if (not skipTest and eval('self.'+plate) == int(value)):
            return

        # Set new variable value
        exec 'self.'+tags[0]

        # New configuration: stop animation
        if not skipTest:
            self.fluidFlow = 0

        kw = {'fill': 'Black', 'font': 'Helvetica 10'}
        self.canvas.itemconfigure(tags[1], **kw)
        kw = {'fill': 'Red', 'font': 'Helvetica 11 bold'}
        self.canvas.itemconfigure(tags[0], **kw)

        if (int(value) and self.plateMovement.get()):
            thread.start_new_thread(self.move_plate, (tags[0], plate, value))

    # Visualize moving plate by a moving arrow
    def move_plate(self, tag, plate, value):
        x0 = 200 - int(value)*118
        x1 = x0
        if plate == 'top':
            y0 = 80 + int(value)*self.tilt*24
        else:
            y0 = 190 + int(value)*self.tilt*24
        length = int(value)*240
        height = -int(value)*self.tilt*48

        while (self.plateMovement.get() and eval('self.'+plate) == int(value)):
            x1 = x1 + int(value)
            if (x1 > x0+int(value)*length or x1 < x0-int(value)*length):
```

```
            x1 = x0+5
        y1 = y0 + height*float(x1-x0)/length

        kw = {'arrow': 'last', 'width': 3, 'tags': 'moving_'+tag}
        self.canvas.create_line(x0,y0, x1,y1, **kw)

        root.update()
        time.sleep(.01)
        self.canvas.delete('moving_'+tag)

# Toggle the pressure
def toggle_pressure(self, event, tag):
    # New configuration: stop animation
    self.fluidFlow = 0

    # Turn pressure off
    if eval('self.'+tag):
        exec 'self.'+tag+'=0'
        kw = {'text': 'Low\nPressure', 'fill': 'Black'}
        self.canvas.itemconfigure(tag, **kw)

    # Turn pressure on
    else:
        exec 'self.'+tag+'=1'
        kw = {'text': 'High\nPressure', 'fill': 'Red'}
        self.canvas.itemconfigure(tag, **kw)

# Start a new thread to visualize fluid flow (or stop active thread)
def start_flow(self):
    # Stop active thread
    if self.fluidFlow:
        self.fluidFlow = 0
        self.startBtn.configure(text = 'Start fluid flow')

    # Start new thread
    else:
        self.fluidFlow = 1
        self.startBtn.configure(text = 'Stop fluid flow')

        n = int(self.counter.get())
        args = (n, self.left, self.right, self.tilt, self.top, self.bottom)
        profile = self.system.compute_flow_profile(*args)
        thread.start_new_thread(self.flow, (profile,))

# The fluid flow visualization (moving arrows)
def flow(self, profile):
    if profile == 'No flow':
        kw = {'text': 'No flow', 'font': 'Helvetica 16 roman',
              'fill': 'DarkGreen', 'tags': 'No_flow'}
        while self.fluidFlow:
            self.canvas.create_text(200, 135, **kw)
            root.update()
            time.sleep(0.4)

            self.canvas.delete('No_flow')
            root.update()
            time.sleep(0.4)

    else:
```

```
            n = len(profile)                    # n = number of arrows
            arrowshapes = ((10,12,4),(8,10,3),(5,7,2))
            shape = arrowshapes[0]              # Large arrowshape
            if n > 12:
                shape=arrowshapes[1]            # Medium arrowshape
            if n > 18:
                shape=arrowshapes[2]            # Small arrowshape

            tilt = self.tilt*25
            while 1:
                kw = {'arrow': 'last', 'arrowshape': shape,
                      'fill': 'DarkGreen', 'tags': 'arrows'}
                for i in range(125+abs(profile[0])*125):
                    for j in range(1,n):
                        x0 = 200 + profile[0]*125
                        x1 = x0 + profile[j]*i
                        y0 = 185 - j*100/n - profile[0]*tilt
                        y1 = y0 - profile[j]*i*2*tilt/250.0
                        self.canvas.create_line(x0, y0, x1, y1, **kw)
                    root.update()
                    time.sleep(0.01)
                    if not self.fluidFlow:
                        return
                    self.canvas.delete('arrows')

    # Launch the system properties window
    def properties(self):
        self.clean_up()
        self.system.prefs.launch_window()

    # Delete flow lines from canvas
    def delete_flow_lines(self):
        self.canvas.delete('arrows')

    # Clean up and wait for threads to die
    def clean_up(self, event=None):
        self.fluidFlow = 0              # Stop the fluid flow
        self.plateMovement.set('0')     # Stop the plate movement animation
        time.sleep(.2)                  # Let threads die first

    # Display the reference page
    def help(self, event = None):
        refPage = 'applications_fluidflow.html'
        if os.name == 'nt':
            app = os.path.normpath('../doc/' + refPage)
        else:
            app = 'netscape ' + os.path.normpath('../doc/' + refPage) + '&'
        thread.start_new_thread(os.system, (app,))

    # Exit program
    def exit(self, event = None):
        self.clean_up()
        root.destroy()
#----


class System:
    def __init__(self):
        self.prefs = Prefs()
```

```
        # The flow profile solution
        self.solution = '-((beta + ro*g*sin(theta))/2*mu)*z*z +'+\
                        '((Uh-U0)/H + (beta +'+\
                        'ro*g*sin(theta))*H/2*mu)*z + U0'

    # Compute the flow profile (n: number of arrows)
    def compute_flow_profile(self, n, left, right, tilt, top, bottom):
        mu = self.prefs.mu
        ro = self.prefs.ro
        g = self.prefs.g
        H = self.prefs.H

        # The gui-changable values:
        beta = self.prefs.beta*(left-right)
        theta = -self.prefs.theta*tilt
        Uh = self.prefs.Uh*top
        U0 = self.prefs.U0*bottom

        # Calculate the z-value for each arrow and normalise: -1<=z<=1
        profile = []
        for i in range(1, n+1):
            z = self.prefs.H*i/float(n+1)
            profile.append(eval(self.solution))
        self.normal = max(abs(max(profile)), abs(min(profile)))
        if self.normal == 0:
            return 'No flow'
        profile = map(self.normalise, profile)

        # Calculate where (x-coordinate) arrows should start
        a = max(profile)
        b = min(profile)
        if misc.sign(a) == misc.sign(b):
            position = -misc.sign(a)
        else:
            position = 1.0-a*2.0/(abs(a)+abs(b))
        profile.insert(0, position)

        return profile

    # normalise profile elements
    def normalise(self, element):
        return element/self.normal
#----


class Prefs(dialogs.BasePrefs):
    """Create a standard preference dialog"""

    def __init__(self):
        dialogs.BasePrefs.__init__(self)

        self.mu = 2.0         # Fluid property (viscosity)
        self.ro = 1.0         # Fluid property (density)
        self.g = 9.8          # Acceleration due to gravity
        self.H = 8.0          # Height (distance between plates)
        self.beta = 1.0       # Strength of pressure
        self.theta = 0.2      # Plate inclination (0,2 ~ pi/16)
        self.Uh = 15          # Speed of upper plate
```

```
        self.U0 = 15          # Speed of lower plate

    def launch_window(self):
        self.create_dialog(root, width = 260, height = 306,
                buttons = ('Cancel', 'Reset', 'Ok'), defaultbutton = 2)

        list = (('Fluid viscosity (mu): ', self.mu, 'real', 0.0, 5.0, ''),
                ('Fluid density (ro): ', self.ro, 'real', 0.0, 5.0, ''),
                ('Gravity (g): ', self.g, 'real', 0.0, 20.0, ''),
                ('Height (H): ', self.H, 'real', 0.5, 20.0, ''),
                ('Pressure strength (beta): ', self.beta, 'real', 0.0, 5.0, ''),
                ('Plate inclination (theta): ', self.theta, 'real', 0.0, pi/2, ''),
                ('Upper plate speed (Uh): ', self.Uh, 'integer', 0, 25, ''),
                ('Lower plate speed(U0): ', self.U0, 'integer', 0, 25, ''))
        self.counters = self.create_counters(list)

    def close_window(self, button):
        if button == 'Reset':
            values = (2.0, 1.0, 9.8, 8.0, 1.0, 0.2, 15, 15)
            for n in range(len(self.counters)):
                self.counters[n].setentry(values[n])
        else:
            if button == 'OK':
                self.mu = float(self.counters[0].get())
                self.ro = float(self.counters[1].get())
                self.g = float(self.counters[2].get())
                self.H = float(self.counters[3].get())
                self.beta = float(self.counters[4].get())
                self.theta = float(self.counters[5].get())
                self.Uh = int(self.counters[6].get())
                self.U0 = int(self.counters[7].get())
            self.dialog.destroy()
#----

root = Tkinter.Tk()
root.title('Fluid flow between plates')

if os.name in ('nt', 'posix'):
    root.option_readfile('fontsAndColors.txt')
else:
    Pmw.initialise(root, fontScheme = 'pmw1')

Gui()

root.resizable(0,0)
root.mainloop()
```

Chapter 13

A visual demonstration of the matrix product (`matrix`)

13.1 Why create a demonstration of the matrix product?

A lot of students, seeing the matrix product for the first time, tend to miss the structure of the method. It usually takes a little time to understand it fully. It is simple to multiply two 3×3 matrices, but not everybody knows how to multiply a 3×1 matrix and a 1×4 matrix. This application will give the user the opportunity to try out different combinations of matrices and see a step by step demonstration of how the product is calculated.

13.2 The illustration

Let the matrix product be given as $A * B = C$. The general idea behind this illustration is quite simple:

- Draw the matrices A and B with some random elements. Let the matrix C be empty.
- Highlight the active row in A and the active column in B.
- Move the elements to be multiplied out of the matrices and calculate the corresponding element in C.
- Move the new element to its position in the C matrix.
- Repeat the process until the C matrix is full.

13.3 The user interface

The GUI in this application is very simple. A *Tkinter.Canvas* to draw and move the matrix elements, two *Pmw.OptionMenus* to choose the matrix sizes, and a row of buttons.

The maximum size of the matrices are limited to 4×4 to reduce the canvas space needed. An error message is shown if two illegal matrix sizes are choosen (for example $A = 2 \times 2$ and $B = 3 \times 2$).

The buttons:

- "Redraw matrices" - Put new random elements into the matrices A and B.
- "Start animation" - Animate the rest of the matrix product.
- "Step by step" - Execute the next step in the matrix product.
- "Preferences" - Display preference window.
- "Help" - Display reference page (a *html* version of this section).
- "Exit" - Quit application.

The `programflow` module (section 9) is used to manage the animation/step by step methods.

The preferences:

- "Element min" - The minimum value of a matrix element. This is an integer in the range [-99, 99].
- "Element max" - The maximum value of a matrix element. This is an integer in the range [-99, 99].
- "Movement speed" - Movement speed of elements. This is an integer in the range [1, 10] where 1 corresponds to the fastest speed. See the `programflow` module (section 9) for more information on this feature.
- "Pause 1" - The pause (in seconds) after two elements from A and B have moved and multiplied. This is a floating-point number in the range [0.0, 10.0].
- "Pause 2" - The pause (in seconds) after a new element in the C matrix has been created. This is a floating-point number in the range [0.0, 10.0].

13.4 Code

```
#!/bin/sh
""":"
exec python $0 ${1+"$@"}
"""

# Make sure the modules are found
import sys
sys.path[:0] = ['../']

# This will enable the program to be run from idle
import os
dirname,basename = os.path.split(sys.argv[0])
os.chdir(os.path.normpath(dirname))

import Tkinter
import Pmw
import whrandom
import time
```

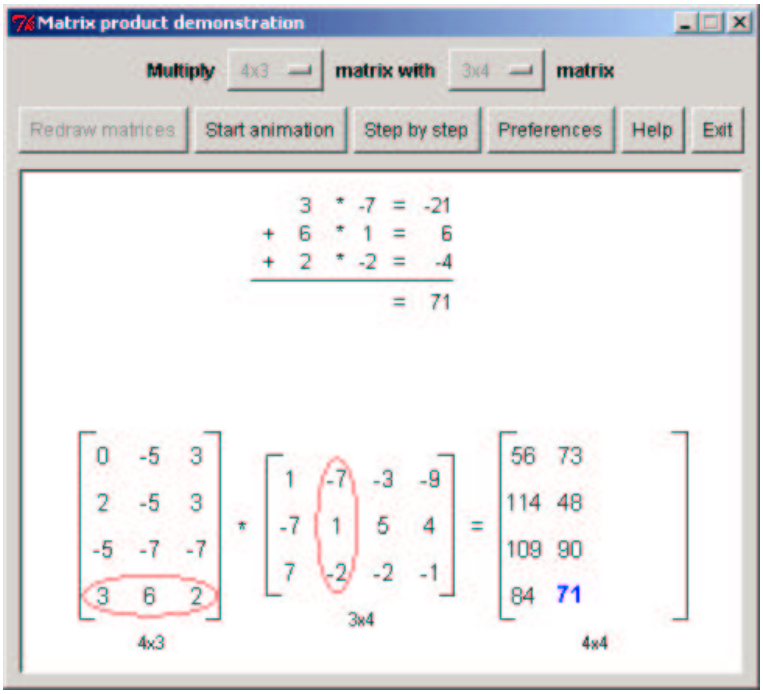


Figure 13.1: Animating the matrix product.

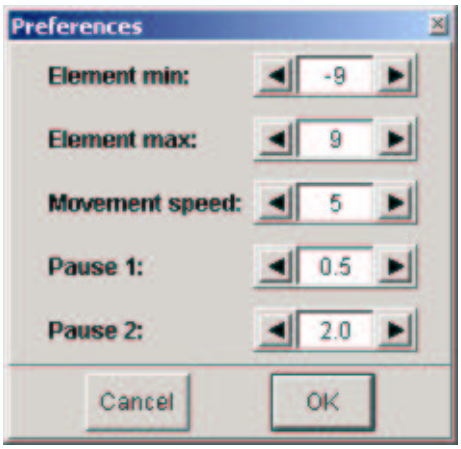


Figure 13.2: The preference window.

```
import thread
from modules import misc
from modules import dialogs
from modules import programflow


class Gui:
    """Handle interface and user interaction"""

    def __init__(self):
        self.bindings()
        self.balloon = Pmw.Balloon(root, initwait = 500)

        # Let class System handle the algorithm
        self.system = System(self)

        # The available matrix sizes
        sizes = ('1x1','1x2','1x3','1x4', '2x1','2x2','2x3','2x4',
                 '3x1','3x2','3x3','3x4', '4x1','4x2','4x3','4x4')

        # Create the matrix size option menus
        frame = Tkinter.Frame(root)
        kw = {'command': self.new_matrix_sizes,
              'labelpos': 'w',
              'label_text': 'Multiply ',
              'menubutton_width': 3,
              'initialitem': '2x3',
              'items': sizes}
        self.matrixAOM = Pmw.OptionMenu(frame, **kw)
        kw['label_text'] = ' matrix with '
        kw['initialitem'] = '3x1'
        self.matrixBOM = Pmw.OptionMenu(frame, **kw)
        label = Tkinter.Label(frame, text = ' matrix')

        # Create the row of buttons below the option menus
        self.buttons = []
        if os.name == 'nt':
            padx = 2
        else:
            padx = 0
        self.buttonBox = Pmw.ButtonBox(root, padx = padx)
        self.buttons.append(self.buttonBox.add('Redraw matrices',
                command = self.new_matrix_sizes))
        self.buttons.append(self.buttonBox.add('Start animation',
                command = self.system.start_animation))
        self.buttons.append(self.buttonBox.add('Step by step',
                command = self.system.flow.next_step))
        self.prefsBtn = self.buttonBox.add('Preferences',
                command = self.launch_prefs_window)
        self.buttonBox.add('Help', command = self.help)
        self.buttonBox.add('Exit', command = self.exit)
        self.buttons.extend((self.matrixAOM.component('menubutton'),
                             self.matrixBOM.component('menubutton')))

        # Create the canvas
        self.canvas = Tkinter.Canvas(root,
                background = 'White',
                borderwidth = 2,
                height = 320,
```

```
            relief = 'sunken',
            width = 460)

    # Position the widgets
    frame.pack(padx = 5, pady = 5)
    self.matrixAOM.pack(side = 'left')
    self.matrixBOM.pack(side = 'left')
    label.pack(side = 'left')
    self.buttonBox.pack(padx = 5)
    self.canvas.pack(padx = 5, pady = 5)


# Create keyboard bindings
def bindings(self):
    root.bind('<q>', self.exit)
    root.bind('<h>', self.help)


# A change in matrix sizes (check for illegal matrix choices)
def new_matrix_sizes(self, choice = None):
    sizeA = self.matrixAOM.getcurselection()
    sizeB = self.matrixBOM.getcurselection()
    if not sizeA[2] == sizeB[0]:
        self.canvas.delete('all')
        kw = {'text': 'Illegal matrix choice!', 'fill': 'Red',
              'font': 'Helvetica 12 bold', 'tags': 'illegal'}
        self.canvas.create_text(230, 140, **kw)
        self.disable_buttons(notOptionMenus = 1)

    else:
        self.canvas.delete('illegal')
        self.enable_buttons()
        self.system.create_matrices(sizeA, sizeB)


# Move elements from matrix A and matrix B and multiply them
def move_elements(self, elementA, elementB, pos):
    y = 25 + pos*18
    kw = {'anchor': 'e', 'font': 'Helvetica 10', 'tags': 'calc'}

    # Do not draw '+'-sign on the first line
    if pos:
        self.canvas.create_text(165, y, text = '+', **kw)

    elementA.move_to(185, y)
    self.canvas.create_text(210, y, text = '*', **kw)

    elementB.move_to(225, y)
    self.canvas.create_text(250, y, text = '=', **kw)

    value = str(elementA.value*elementB.value)
    self.canvas.create_text(280, y, text = value, **kw)

    return int(value)


# Draw the new element in matrix C
def draw_new_calculated_element(self, pos, sum):
    y = 5 + pos*18
```

```
        kw = {'anchor': 'e', 'font': 'Helvetica 10', 'tags': 'calc'}
        self.canvas.create_line(150, y+14, 280, y+14, tags = 'calc')
        self.canvas.create_text(250, y+28, text = '=', **kw)
        self.canvas.create_text(280, y+28, text = str(sum), **kw)
        return 280,y+28


    # Create finished message in canvas
    def create_finished_message(self):
        kw = {'text': 'Finished!', 'fill': 'Red', 'font': 'Helvetica 11 bold',
              'tags': 'finished'}
        self.canvas.create_text(230, 135, **kw)


    # Disable the buttons: Redraw matrices, Start animation, Step by step
    # and the OptionMenus
    def disable_buttons(self, notOptionMenus = 0):
        buttons = self.buttons
        if notOptionMenus:
            buttons = self.buttons[:-2]
        for button in buttons:
            button.configure(state = 'disabled')


    # Enable the buttons: Redraw matrices, Start animation, and Step by step
    # and the OptionMenus
    def enable_buttons(self):
        for button in self.buttons:
            button.configure(state = 'normal')


    # Launch the preferences window
    def launch_prefs_window(self):
        self.prefsBtn.configure(state = 'disabled')
        prefs.launch_window()


    # Display the reference page
    def help(self, event = None):
        refPage = 'applications_matrix.html'
        if os.name == 'nt':
            app = os.path.normpath('../doc/' + refPage)
        else:
            app = 'netscape ' + os.path.normpath('../doc/' + refPage) + '&'
        thread.start_new_thread(os.system, (app,))


    # Exit program
    def exit(self, event = None):
        root.destroy()
#----


class System:
    """Handle the algorithm"""

    def __init__(self, gui):
        self.gui = gui
```

```
        # The matrices in A*B=C (Initialise them later)
        self.matrixA = None
        self.matrixB = None
        self.matrixC = None

        # Use module ProgramFlow to handle algorithm flow
        self.flow = programflow.ProgramFlow(root, self.matrix_product,
                speedFactor = 1500)


    # Automate the rest of the steps
    def start_animation(self):
        self.gui.disable_buttons()
        self.flow.start_animation()


    # This is the actual algorithm for the matrix product
    def matrix_product(self):
        self.gui.canvas.delete('finished')
        self.matrixC.delete_all_elements()
        for i in (0, 3, 4):
            self.gui.buttons[i].configure(state = 'disabled')

        sum = 0
        for column in range(self.matrixB.columns):
            self.matrixB.highlight_column(column)

            for row in range(self.matrixA.rows):
                self.matrixA.highlight_row(row)
                self.gui.canvas.delete('calc')

                for pos in range(self.matrixB.rows):
                    elementA = self.matrixA.elements[row][pos]
                    elementB = self.matrixB.elements[pos][column]

                    # Highlight new elements
                    elementA.highlight()
                    elementB.highlight()

                    # Move and multiply elements
                    value = self.gui.move_elements(elementA, elementB, pos)
                    sum = sum + value

                    self.flow.wait_or_next_step(seconds = prefs.pause1)
                    elementA.remove_highlight()
                    elementB.remove_highlight()

                # Create new element and move it to C-matrix
                x,y = self.gui.draw_new_calculated_element(pos+1, sum)
                elementC = self.matrixC.create_element(x,y, row, column, sum)

                if not self._last_element(row, column, pos):
                    self.flow.wait_or_next_step(seconds = prefs.pause2)
                    elementC.remove_highlight()

        # Clean up
        self.flow.animation = 0
        self.flow.started = 0
        self.gui.create_finished_message()
```

```
        self.gui.enable_buttons()


    # Create and draw the matrices A, B and C (do not fill the elements in C)
    def create_matrices(self, sizeA, sizeB):
        # Clean up
        self.gui.canvas.delete('all')

        # Create matrix A
        columns = int(sizeA[2]) + 2*int(sizeB[2])
        x = 230 - columns*15 - 10
        y = 230 - (int(sizeA[0]) - 1)*15
        self.matrixA = Matrix(x,y, int(sizeA[0]), int(sizeA[2]),
                self.gui.canvas)

        # Create the center dot
        x = x + (int(sizeA[2]) - 1)*30 + 30
        y = 230 + 4
        self.gui.canvas.create_text(x,y, text = '*', font = 'Helvetica 12')

        # Create matrix B
        x = x + 30
        y = 230 - (int(sizeB[0]) - 1)*15
        self.matrixB = Matrix(x,y, int(sizeB[0]), int(sizeB[2]),
                self.gui.canvas)

        # Create the equal sign
        x = x + (int(sizeB[2]) - 1)*30 + 30
        y = 230
        self.gui.canvas.create_text(x,y, text = '=', font = 'Helvetica 12')

        # Create matrix C (without elements)
        x = x + 30
        y = 230 - (int(sizeA[0]) - 1)*15
        self.matrixC = Matrix(x,y, int(sizeA[0]), int(sizeB[2]),
                self.gui.canvas, 1)


    # A test to determine if the last element has been processed
    def _last_element(self, row, column, pos):
        return (row     ==  self.matrixA.rows - 1 and
                column  ==  self.matrixB.columns - 1 and
                pos     ==  self.matrixB.rows - 1)
#----


class Matrix:
    def __init__(self, x,y, rows, columns, canvas, nodraw = 0):
        self.x = x              # Position of first element (x-canvas)
        self.y = y              # Position of first element (y-canvas)
        self.rows = rows        # The number of rows in the matrix
        self.columns = columns  # The number of columns in the matrix
        self.canvas = canvas
        self.elements = []

        self.draw(nodraw)

    # Draw the matrix after initialization
    def draw(self, nodraw):
```

```
        # Draw the [] of the matrix
        coords = (self.x-5,     self.y-15,
                  self.x-15,    self.y-15,
                  self.x-15,    self.y+(self.rows-1)*30+15,
                  self.x-5,     self.y+(self.rows-1)*30+15)
        self.canvas.create_line(coords)
        coords = (self.x+(self.columns-1)*30+5,     self.y-15,
                  self.x+(self.columns-1)*30+15,    self.y-15,
                  self.x+(self.columns-1)*30+15,    self.y+(self.rows-1)*30+15,
                  self.x+(self.columns-1)*30+5,     self.y+(self.rows-1)*30+15)
        self.canvas.create_line(coords)

        # Draw the subtitle of the matrix (rows x columns)
        coords = (self.x+(self.columns-1)*15, self.y+self.rows*30)
        text = str(self.rows) + 'x' + str(self.columns)
        self.canvas.create_text(coords, text = text)

        # Create and draw the elements if not nodraw=1 (only matrix C)
        if nodraw:
            return
        for y in range(self.rows):
            self.elements.append([])
            for x in range(self.columns):
                element = Element(x, y, self.x+x*30, self.y+y*30, self.canvas)
                self.elements[y].append(element)


    # Create an element and move it to position [column,row] (only matrix C)
    def create_element(self, x, y, row, column, value):
        element = Element(row, column, x, y, self.canvas, value)
        self.elements.append(element)
        element.highlight()
        element.move_to(self.x+column*30, self.y+row*30, makeCopy = 0)
        return element


    # Delete all elements (only matrix C)
    def delete_all_elements(self):
        for element in self.elements:
            self.canvas.delete(element.tag)
        self.elements = []


    # Highlight row number n (only matrix A)
    def highlight_row(self, n):
        self.canvas.delete('highlight_row')
        x = self.x + (self.columns - 1)*15
        y = self.y + n*30
        dx = self.columns*15 - 2
        dy = 13
        kw = {'outline': prefs.colour1, 'tags': 'highlight_row'}
        self.canvas.create_oval(x-dx,y-dy, x+dx,y+dy, **kw)


    # Highlight column number n (only matrix B)
    def highlight_column(self, n):
        self.canvas.delete('highlight_column')
        x = self.x + n*30
        y = self.y + (self.rows - 1)*15
```

```
        dx = 13
        dy = self.rows*15 - 2
        kw = {'outline': prefs.colour1, 'tags': 'highlight_column'}
        self.canvas.create_oval(x-dx,y-dy, x+dx,y+dy, **kw)
#----


class Element:
    def __init__(self, row, column, x, y, canvas, value = 'no'):
        self.row = row        # Position (row in matrix)
        self.column = column  # Position (column in matrix)
        self.x = x            # Position (x-canvas coordinate)
        self.y = y            # Position (y-canvas coordiante)
        self.canvas = canvas

        if value == 'no':
            self.value = whrandom.randint(prefs.min, prefs.max)
        else:
            self.value = value

        self.tag = str(x) + '_' + str(y) + '_' + str(row) + '_' + str(column)
        kw = {'text': str(self.value), 'font': 'Helvetica 11', 'tag': self.tag}
        self.canvas.create_text(x,y, **kw)


    # Make a copy (if not called from matrix C) and move it to x,y
    def move_to(self, x, y, makeCopy = 1):
        if makeCopy:
            tag = self.tag + 'copy'
            kw = {'text': str(self.value), 'font': 'Helvetica 10',
                  'tag': ('calc', tag)}
            self.canvas.create_text(self.x, self.y, **kw)
        else:
            tag = self.tag

        path = misc.create_path(self.x, self.y, x, y)
        for dx,dy in path:
            self.canvas.move(tag, dx, dy)
            gui.system.flow.update_and_pause(speed = prefs.speed)

    def highlight(self):
        kw = {'fill': prefs.colour2, 'font': 'Helvetica 11 bold'}
        self.canvas.itemconfigure(self.tag, **kw)

    def remove_highlight(self):
        kw = {'fill': 'Black', 'font': 'Helvetica 11 roman'}
        self.canvas.itemconfigure(self.tag, **kw)
#----


class Prefs(dialogs.BasePrefs):
    """Create standard preference dialog"""

    def __init__(self, gui):
        dialogs.BasePrefs.__init__(self, gui.balloon)

        self.min = -9         # Minimum value of matrix elements
        self.max = 9          # Maximum value of matrix elements
        self.speed = 5        # 1:fast, 10:slow
```

```
        self.pause1 = .50     # Pause after each element move (in seconds)
        self.pause2 = 2.00    # Pause after each new element made (in seconds)
        self.colour1 = 'Red'  # Highlight colour row/column
        self.colour2 = 'Blue' # Highlight colour matrix element


    def launch_window(self):
        self.create_dialog(root, width = 230, height = 206)

        list = (('Element min: ', self.min, 'integer', -99, 99,
                 'Minimum value of a matrix element'),
                ('Element max: ', self.max, 'integer', -99, 99,
                 'Maximum value of a matrix element'),
                ('Movement speed: ', self.speed, 'integer', 1, 10,
                 'Movement speed of elements (1:fast, 10:slow)'),
                ('Pause 1: ', self.pause1, 'real', 0.0, 10.0,
                 'Pause (in secs) after two elements have multiplied'),
                ('Pause 2: ', self.pause2, 'real', 0.0, 10.0,
                 'Pause (in secs) after a new C-matrix value has been created'))
        self.counters = self.create_counters(list)


    def close_window(self, button):
        if button == 'OK':
            min = int(self.counters[0].get())
            max = int(self.counters[1].get())
            if min > max:
                message = 'Element min is larger\nthan element max'
                self.display_error_message(message, 180, 115)
                return
            self.min = min
            self.max = max
            self.speed = int(self.counters[2].get())
            self.pause1 = float(self.counters[3].get())

            self.pause2 = float(self.counters[4].get())

        self.dialog.destroy()
        gui.prefsBtn.configure(state='normal')
#----

root = Tkinter.Tk()
root.title('Matrix product demonstration')

if os.name in ('nt', 'posix'):
    root.option_readfile('fontsAndColors.txt')
else:
    Pmw.initialise(root, fontScheme = 'pmw1')

gui = Gui()
prefs = Prefs(gui)
gui.new_matrix_sizes()

root.resizable(0,0)
root.mainloop()
```

Chapter 14

Fourth order Runge-Kutta method (rungekutta)

14.1 The numerical mehtod

The fourth order Runge-Kutta method generates numerical solutions to systems of first order intial-value problems written on the form

$$\begin{aligned} x_1' &= f_1(t, x_1, x_2, \ldots), \quad x_1(t_0) = x_{1_0}, \\ x_2' &= f_2(t, x_1, x_2, \ldots), \quad x_2(t_0) = x_{2_0}, \\ &\vdots \end{aligned}$$

It is assumed that the functions $f_1, f_2, \ldots$ are sufficiently smooth so that unique solutions exist. The method approximates solutions $x_1^k, x_2^k, \ldots$ to the system at the points

$$t_k = t_0 + kh, \quad h > 0, \quad k = 0, 1, \ldots$$

and is written as

$$\begin{aligned} x_1^{k+1} &= x_1^k + \frac{1}{6}\big(a1_1^k + 2a2_1^k + 2a3_1^k + a4_1^k\big), \\ x_2^{k+1} &= x_2^k + \frac{1}{6}\big(a1_2^k + 2a2_2^k + 2a3_2^k + a4_2^k\big), \\ &\vdots \end{aligned}$$

where

$$\begin{aligned} a1_i^k &= hf_i\big(t_k, x_1^k, x_2^k, \ldots\big), \\ a2_i^k &= hf_i\big(t_k + \frac{h}{2}, x_1^k + \frac{a1_1^k}{2}, x_2^k + \frac{a1_2^k}{2}, \ldots\big), \\ a3_i^k &= hf_i\big(t_k + \frac{h}{2}, x_1^k + \frac{a2_1^k}{2}, x_2^k + \frac{a2_2^k}{2}, \ldots\big), \\ a4_i^k &= hf_i\big(t_k + h, x_1^k + a3_1^k, x_2^k + a3_2^k, \ldots\big). \end{aligned}$$

The total discretization error is proportional to h^4. More information on the Runge-Kutta method may be found in [9].

14.2 About the default system

The default system has its origin in a second order, linear, nonhomogenous differential equation:

$$\frac{dx^2}{dt^2} + 4x = \cos t, \quad x(0) = x'(0) = 0. \tag{2.1}$$

The equation resembles the equation for the forced vibrations application (section 16.5) except that the spring-mass system now is undamped. To form a system of first order equations, let $x_2' = x_1 = x$. Then

$$\begin{aligned} x_1' &= -4x_2 + \cos t, \quad x_1(0) = 0, \\ x_2' &= x_1, \quad x_2(0) = 0. \end{aligned}$$

The exact solution to (2.1) is

$$x = \frac{1}{3}\big(\cos t - \cos 2t\big).$$

14.3 The illustration

For each new time step in the algorithm:

- Calculate the coefficients.
- Calculate new approximate solutions.
- Update the "Current t:" label.
- Add the new solutions to the graph.
- Add the new solutions to the computed values table if necessary.

14.4 The user interface

The GUI consists of the following elements:

- A *Pmw.MenuBar* (see below).
- A *Pmw.Counter* to set the number of equations in the system. A maximum number of nine is allowed.
- Two *Pmw.EntryFields* used to type in the initial time value (t_0) and the time step (h).
- A *Tkinter.Label* showing the current time step.

- A table of *Pmw.EntryFields*. The number of rows corresponds to the number of equations used for the system. This will change whenever the *Pmw.Counter* changes value. The table has two rows, one for the actual equations and one for initial values. The equations have to be typed in with a Python math type syntax. The variables must be named $x1, x2, \ldots$.
- A row of buttons (see below).
- A *BLT.Graph* widget to display the solutions as functions $x_1(t), x_2(t), \ldots$.

The MenuBar has three buttons:

- File
  - "New system" - Clear entry fields, "Current t:" label, graph widget and computed values table.
  - "Use default system" - Clear widgets (as above) and insert the default system (see section 14.2).
  - "Dump computed values to file" - Save the computed values in a text file.
  - "Exit" - Quit application.
- Options
  - "Show computed values as a table" - Display the computed values table.
  - "Show plot of" - This is a list of checkbuttons used to choose which of the $x_1, x_2, \ldots$ solutions to plot in the graph widget. The default is "all".
  - "Animation speed" - Choose the speed of the animation, i.e. how much to pause between each time step in animation mode. The default is "medium".
  - "Graph symbol" - A list of possible graph node symbols. The default is "none".
- Help
  - "Reference page" - Display the reference page (a *html* version of this section).

The row of buttons:

- "Restart" - Restart the algorithm using the current system of equations. This will not turn off the animation mode.
- "Start/stop animation" - Start or stop animating the algorithm. The speed of the animation is determined by the "Options/Animation speed" menu.
- "Next step" - Execute the next time step in the algorithm.
- "Help" - Display the reference page (a *html* version of this section).
- "Exit" - Quit application.

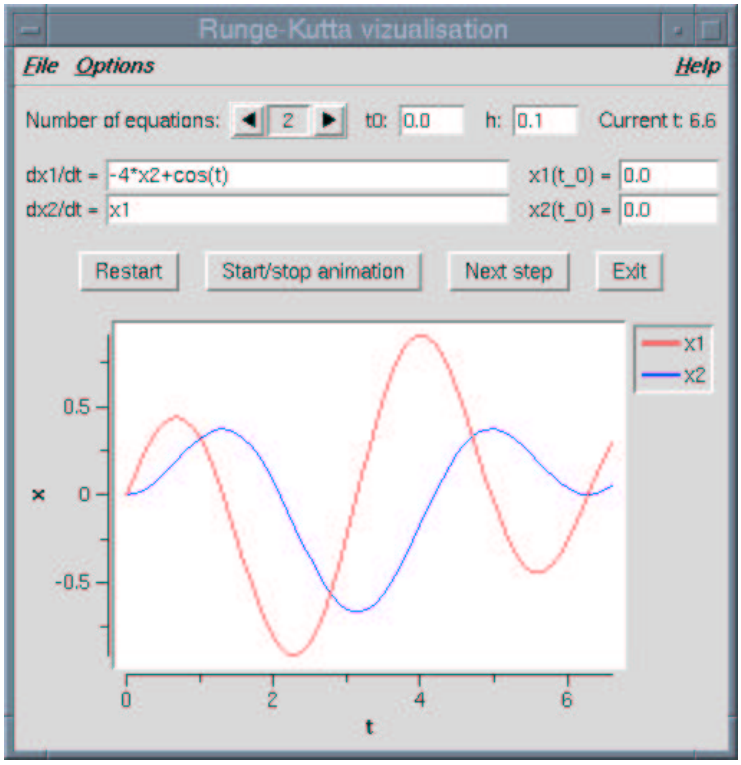


Figure 14.1: Plotting solution to the default system.

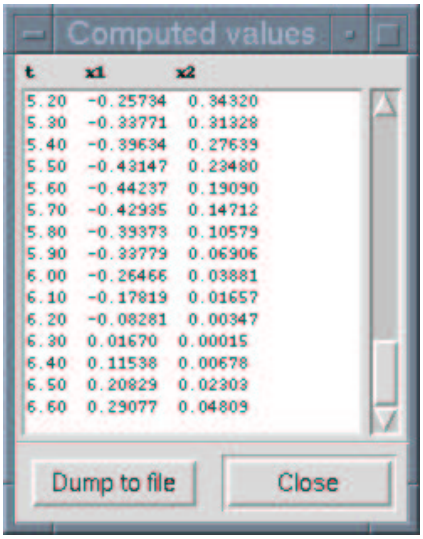


Figure 14.2: The computed values table.

14.5 Code

```
#!/bin/sh
""":"
exec python $0 ${1+"$@"}
"""

# Make sure the modules are found
import sys
sys.path[:0] = ['../']

# This will enable the program to be run from idle
import os
dirname,basename = os.path.split(sys.argv[0])
os.chdir(os.path.normpath(dirname))

import Tkinter
import Pmw
import re
import tkFileDialog
import time
import thread
from modules import misc
from modules import programflow
from modules import widgets
from math import *


class Gui:
    """Handle interface and user interaction"""

    def __init__(self):
        # Create an object to handle the system specific calculations
        self.system = System(self)

        # Some colors for the graph elements
        self.colors = ('red', 'blue', 'green', 'brown', 'yellow',
                       'orange', 'pink', 'black', 'grey')

        # These variables decide which graph elements to show
        # The first one corresponds to all
        self.showGraph = []

        for i in range(10):
            self.showGraph.append(Tkinter.IntVar())
        self.showGraph[0].set(1)

        # Default animation speed (medium) given as the
        # time (in seconds) to pause after each time step
        self.speed = .05
        self.speeds = ['turtle', 'slow', 'medium', 'fast', 'rabbit']

        # Default symbol for graph elements
        self.symbol = 'none'
        self.symbols = ('none', 'square', 'circle', 'diamond', 'plus',
                       'cross', 'splus', 'scross', 'triangle')

        # Create MenuBar and a separator
        self.menuBar = Pmw.MenuBar(root)
```

```
separator = widgets.create_separator(root)

# Create counter widget
validator = {'validator': 'integer', 'min': 2, 'max': 9}
self.eqCounter = Pmw.Counter(root,
        datatype = self.new_counter_value,
        entryfield_validate = validator,
        entryfield_value = '2',
        entry_justify = 'center',
        entry_state = 'disabled',
        entry_width = 3,
        labelpos = 'w',
        label_text = 'Number of equations: ')

# Create EntryFields for t0 and h, and the current t label
self.t0Entry = Pmw.EntryField(root,
        entry_background = 'White',
        entry_width = 5,
        labelpos = 'w',
        label_text = 't0: ',
        validate = {'validator': 'real'})
self.hEntry = Pmw.EntryField(root,
        entry_background = 'White',
        entry_width = 5,
        labelpos = 'w',
        label_text = 'h: ',
        validate = {'validator': 'real', 'min': 0, 'max': 1})
self.tLabel = Tkinter.Label(root,
        text = 'Current t: '+str(self.system.t0))

# Create a Frame to hold the equation/initial value entryfields
self.eqFrame = Tkinter.Frame(root)
self.eqEntries = []

# Create a buttonbox
buttons = Pmw.ButtonBox(root)
buttons.add('Restart', command = self.system.restart)
buttons.add('Start/stop animation',
        command = self.system.start_or_stop)
buttons.add('Next step', command = self.system.next_step)
buttons.add('Help', command = self.help)
buttons.add('Exit', command = self.exit)

# Create a Pmw.Blt window
self.blt = Pmw.Blt.Graph(root,
        height = 300,
        width = 450)
font = 'Helvetica 12 bold'
self.blt.xaxis_configure(title = 't', titlefont = font)
self.blt.yaxis_configure(title = 'x', titlefont = font)

# Position the widgets
self.menuBar.grid(row = 0, columnspan = 4, sticky = 'ew')
separator.grid(row = 1, columnspan = 4, sticky = 'ew')
self.eqCounter.grid(row = 2, sticky = 'w', padx = 5, pady = 10)
self.t0Entry.grid(row = 2, column = 1, padx = 5)
self.hEntry.grid(row = 2, column = 2, padx = 5)
self.tLabel.grid(row = 2, column = 3, padx = 5)
self.eqFrame.grid(row = 3, columnspan = 4, sticky = 'ew', pady = 3)
```

```
        buttons.grid(row = 4, columnspan = 4, padx = 5, pady = 5)
        self.blt.grid(row = 5, columnspan = 4, sticky = 'news', padx = 5)
        root.grid_rowconfigure(6, minsize = 5)

        # Allow (some of) the widgets to expand
        root.grid_columnconfigure(0, weight = 1)
        root.grid_columnconfigure(1, weight = 1)
        root.grid_rowconfigure(5, weight = 1)
        self.eqFrame.grid_columnconfigure(0, weight = 1)

        # Create an object to handle the table of computed values
        self.table = Table(self)

        # Create the entries in the menu amnd configure menu buttons
        buttons = self.create_menu()
        misc.MenuBarConfig(self.menuBar).configure(buttons)

        # Initialise default system
        self.default_system()

        self.bindings()

    # Define keyboard shortcuts
    def bindings(self):
        root.bind('<Control-q>', self.exit)
        root.bind('<Control-h>', self.help)

    # Create the menus in the menubar
    def create_menu(self):
        # Create file menu
        self.menuBar.addmenu('File', '')
        self.menuBar.addmenuitem('File', 'command',
                label = 'New system',
                command = self.new_system)
        self.menuBar.addmenuitem('File', 'command',
                label = 'Use default system',
                command = self.default_system)
        self.menuBar.addmenuitem('File', 'command',
                label = 'Dump computed values to file',
                command = self.table.dump_values_to_file)
        self.menuBar.addmenuitem('File', 'separator')
        self.menuBar.addmenuitem('File', 'command',
                label = 'Exit',
                command = self.exit)

        # Create options menu
        self.menuBar.addmenu('Options', '')
        self.menuBar.addmenuitem('Options', 'command',
                label = 'Show computed values as a table',
                command = self.table.display_dialog)
        self.menuBar.addcascademenu('Options', 'Show plot of')
        self.menuBar.addmenuitem('Show plot of', 'checkbutton',
                 label = 'all',
                 command = misc.Command(self.show_plot_of, 'all'),
                 variable = self.showGraph[0])
        self.menuBar.addcascademenu('Options', 'Animation speed')
        for speed in self.speeds:
            self.menuBar.addmenuitem('Animation speed', 'command',
                    label = speed,
```

```
                command = misc.Command(self.change_speed, speed))
        self.menuBar.addcascademenu('Options', 'Graph symbol')
        for symbol in self.symbols:
            self.menuBar.addmenuitem('Graph symbol', 'command',
                    label = symbol,
                    command = misc.Command(self.change_symbol, symbol))

        # Create help menu
        self.menuBar.addmenu('Help', '', side = 'right')
        self.menuBar.addmenuitem('Help', 'command',
                label = 'Reference page', command = self.help)

        # Return menu buttons
        return ('File', 'Options', 'Help')

    # Handle adding/removing of equation entryfields. This function
    # is called when either of the counter buttons are pressed.
    def new_counter_value(self, text, factor, incr):
        if not ((text == '2' and factor == -1) or
                (text == '9' and factor == 1)):
            if factor == 1:
                self.add_entryfields()
            else:
                self.remove_entryfields()
            self.table.create_label()
            self.update_show_plot_of_list()
        return str(int(text) + factor)

    # Create equation and initial value entryfields
    def add_entryfields(self):
        n = len(self.eqEntries) + 1
        equationEF = self.create_EF(30, 'dx' + str(n) + '/dt =', n, 0)
        initialValueEF = self.create_EF(8, 'x' + str(n) + '(t0) =', n, 1)
        self.eqEntries.append((equationEF, initialValueEF))

    # Remove the last equation and initial value entryfields
    def remove_entryfields(self):
        equationEF,initialValueEF = self.eqEntries.pop()
        equationEF.destroy()
        initialValueEF.destroy()

    # Create, position, and return a Pmw.EntryField
    def create_EF(self, width, text, row, column):
        entryfield = Pmw.EntryField(self.eqFrame,
                entry_background = 'White',
                entry_width = width,
                labelpos = 'w',
                label_text = text)
        entryfield.grid(row = row, column = column, sticky = 'ew', padx = 5)
        return entryfield

    # The new system file menu command has been envoked
    def new_system(self):
        self.system.new_system()
        self.insert_system_values_in_gui()
        self.clear_blt_and_t_label(0.0)
        self.update_show_plot_of_list()
        self.table.create_label()
        self.table.listbox.clear()
```

```
# The default system file menu command has been envoked
def default_system(self):
    self.system.default_system()
    self.insert_system_values_in_gui()
    self.update_show_plot_of_list()
    try:
        self.table.create_label()
        self.table.listbox.clear()
    except:
        pass

# Insert the values defined in the system class into the GUI
def insert_system_values_in_gui(self):
    n = len(self.system.equations)

    # Remove equation entryfields (if too many)
    for i in range(len(self.eqEntries) - n):
        self.remove_entryfields()

    # Add equation entryfields (if too few)
    for i in range(n - len(self.eqEntries)):
        self.add_entryfields()

    # Fill the equation entryfields
    for i in range(n):
        equationEF,initialValueEF = self.eqEntries[i]
        equationEF.setentry(self.system.equations[i])
        initialValueEF.setentry(str(self.system.initialValues[i]))
    self.eqCounter.component('entryfield').setentry(str(n))
    self.t0Entry.setentry(str(self.system.t0))
    self.hEntry.setentry(str(self.system.h))
    self.tLabel.configure(text = 'Current t: ' + str(self.system.t0))

# Set new animation speed
def change_speed(self, speed):
    index = self.speeds.index(speed)
    self.speed = .2 - index*0.05

# Change the symbol of all blt elements
def change_symbol(self, symbol):
    self.symbol = symbol
    for name in self.blt.element_names():
        self.blt.element_configure(name, symbol = symbol)

# Delete the blt elements and reset the tLabel
def clear_blt_and_t_label(self, t0):
    for name in self.blt.element_names():
        self.blt.element_delete(name)
    self.tLabel.configure(text = 'Current t: ' + str(t0))

# Update the t-label, table, and graph
def display_solution(self, tdata, xdata):
    self.tLabel.configure(text = 'Current t: ' + str(tdata[-1]))
    self.table.insert_solution(tdata[-1], xdata[-1])
    self.draw_graph_solution(tdata, xdata)

# Update the graph representation of the solution
def draw_graph_solution(self, tdata, xdata):
```

```
        for i in range(len(self.system.equations)):
            name = 'x' + str(i+1)
            if not self.blt.element_exists(name):
                self.blt.line_create(name, symbol = self.symbol,
                        color = self.colors[i])

            # Find the correct x-values for this graph
            x = []
            for xlist in xdata:
                x.append(xlist[i])

            # Update graph data
            self.blt.element_configure(name, xdata = tuple(tdata))
            self.blt.element_configure(name, ydata = tuple(x))

    # Update the menu checkbuttons 'Show plot of'
    def update_show_plot_of_list(self):
        self.menuBar.deletemenuitems('Show plot of', 0, 9)
        self.menuBar.addmenuitem('Show plot of', 'checkbutton',
                 label = 'all',
                 command = misc.Command(self.show_plot_of, 'all'),
                 variable = self.showGraph[0])
        for i in range(len(self.eqEntries)):
            name = 'x' + str(i+1)
            self.menuBar.addmenuitem('Show plot of', 'checkbutton',
                    label = name,
                    command = misc.Command(self.show_plot_of, name),
                    variable = self.showGraph[i+1])

    # Change which graphs to display
    def show_plot_of(self, name):
        if name == 'all':
            self.showGraph[0].set(1)
        else:
            self.showGraph[0].set(0)
        for i in range(len(self.system.equations)):
            element = 'x' + str(i+1)
            if self.blt.element_exists(element):
                if name == 'all':
                    self.blt.element_configure(element, hide = 0)
                    self.showGraph[i+1].set(0)
                elif name == element:
                    self.blt.element_configure(element, hide = 0)
                    self.showGraph[i+1].set(1)
                else:
                    self.blt.element_configure(element, hide = 1)
                    self.showGraph[i+1].set(0)

    # Display reference page
    def help(self, event = None):
        refPage = 'applications_rungekutta.html'
        if os.name == 'nt':
            app = os.path.normpath('../doc/' + refPage)
        else:
            app = 'netscape ' + os.path.normpath('../doc/' + refPage) + '&'
        thread.start_new_thread(os.system, (app,))

    # Exit program
    def exit(self, event = None):
```

```
        root.destroy()
#----


class Table:
    """Handle the computed values table"""

    def __init__(self, gui):
        self.gui = gui

        # Create a separate dialog window
        self.dialog = Pmw.Dialog(root,
                buttons = ('Dump to file', 'Close'),
                command = self.process_buttons,
                defaultbutton = 1,
                title = 'Computed values')
        self.dialog.withdraw()
        self.dialog.resizable(0,0)

        # Enable table stretching
        self.dialog.interior().grid_rowconfigure(0, weight = 1)
        self.dialog.interior().grid_columnconfigure(0, weight = 1)

        # Create a listbox to display the table
        # Use a fixed-size font
        if os.name == 'nt':
            font = 'Courier 9'
        else:
            font = 'Courier 10'
        self.listbox = Pmw.ScrolledListBox(self.dialog.interior(),
                hull_height = 225,
                hull_width = 200,
                labelpos = 'nw',
                label_font = font + ' bold',
                listbox_background = 'White',
                listbox_font = font,
                usehullsize = 1,
                vscrollmode = 'static')
        self.listbox.grid(padx = 2, pady = 2, sticky = 'news')
        self.create_label()

    # Display the table dialog
    def display_dialog(self):
        self.dialog.deiconify()

    # The label will change if the number of equations change
    def create_label(self):
        label = 't     x1       x2'
        n = len(self.gui.eqEntries)
        for i in range(2,n):
            label = label + '       ' + 'x' + str(i+1)
        self.listbox.configure(label_text = label)
        self.listbox.configure(hull_width = 70 + n*65)

    # Add the solution to the listbox
    def insert_solution(self, t, solution):
        text = '%(t)-2.2f' %vars()
        for x in solution:
            text = text + '  %(x)-5.5f' %vars()
```

```
        self.listbox.insert('end', text)
        self.listbox.see('end')

    # Decide what to do when one of the button has been pressed
    def process_buttons(self, button):
        if button == 'Dump to file':
            self.dump_values_to_file()
        else:
            self.dialog.withdraw()

    # Dump the contents of the table to file
    def dump_values_to_file(self):
        filename = tkFileDialog.asksaveasfilename(
                initialfile = 'results.txt')
        if not filename:
            return
        try:
            file = open(filename, 'w')
            for values in self.listbox.get():
                file.write(values+'\n')
        except:
            print 'Couldn\'t write to file',filename
#----


class System:
    """Handle system specific calculations"""

    def __init__(self, gui):
        self.gui = gui
        self.newSystem = 1

        self.equations = []      # The equations as texts
        self.initialValues = []  # The initial values as numbers

        self.t0 = 0.0   # Initial time value
        self.h = 0.1    # Step size (t = t + h)

        # Let the ProgramFlow class handle the flow of the algorithm
        self.flow = programflow.ProgramFlow(root, self.rungekutta)

    # Restart the algorithm
    def restart(self):
        self.gui.table.listbox.clear()
        self.gui.clear_blt_and_t_label(self.t0)
        self.flow.restart()

    # Start or stop (pause) the algorithm
    def start_or_stop(self):
        if self.newSystem:
            if not self.get_new_values():
                return
        self.newSystem = 0
        if self.flow.animation:
            self.flow.stop_animation()
        else:
            self.flow.start_animation()

    # Get entryfield values and reset system texts
```

```
def get_new_values(self):
    self.equations = []
    self.initialValues = []
    try:
        self.t0 = float(self.gui.t0Entry.get())
        self.h = float(self.gui.hEntry.get())
    except:
        return 0
    for equation,initvalue in self.gui.eqEntries:
        if not equation.get():
            return 0
        self.equations.append(str(equation.get()))
        try:
            self.initialValues.append(float(initvalue.get()))
        except:
            return 0
    if len(self.equations) < 1:
        return 0
    else:
        return 1

# Do the next step in the algorithm
def next_step(self):
    self.flow.next_step()

# Remove system values
def new_system(self):
    self.newSystem = 1
    self.t0 = 0.0
    self.h0 = 0.1
    for i in range(len(self.equations)):
        self.equations[i] = ''
        self.initialValues[i] = 0.0
    self.flow.started = 0

# Set default system values
def default_system(self):
    self.t0 = 0.0
    self.h = 0.1
    self.equations = ['-4*x2+cos(t)', 'x1']
    self.initialValues = [0.0, 0.0]
    self.flow.started = 0

# This is the actual algorithm
def rungekutta(self):
    tdata = [self.t0,]
    xdata = [self.initialValues[:],]

    while self.flow.started:
        t = tdata[-1]
        oldX = xdata[-1]
        a1,a2,a3,a4 = self.coefficients(self.equations, t, oldX, self.h)
        tdata.append(t + self.h)
        newX = []
        for i in range(len(self.equations)):
            x = oldX[i] + (a1[i] + 2*a2[i] + 2*a3[i] + a4[i])/6.0
            newX.append(x)
        xdata.append(newX)
        self.gui.display_solution(tdata, xdata)
```

```
            self.flow.wait_or_next_step(seconds = self.gui.speed)

    # Compute and return the coefficients in the rungekutta algorithm
    def coefficients(self, functions, t, x, h):
        length = len(functions)

        # a1[i] = h*f[i](t, x1, x2, ...)
        a1 = []
        for i in range(length):
            func = re.sub('t', str(t), functions[i])    # substitute t
            for n in range(length):                     # run through all x
                xnr = 'x' + str(n+1)
                func = re.sub(xnr, str(x[n]), func)     # substitute x
            a1.append(h*eval(func))                     # evaluate a1

        # a2[i] = h*f[i](t+h/2, x1+a1[1]/2, x2+a1[2]/2, ...)
        a2 = []
        for i in range(length):
            func = re.sub('t', str(t+h/2.0), functions[i])
            for n in range(length):
                xnr = 'x' + str(n+1)
                func = re.sub(xnr, str(x[n]+a1[n]/2.0), func)
            a2.append(h*eval(func))

        # a3[i] = h*f[i](t+h/2, x1+a2[1]/2, x2+a2[2]/2, ...)
        a3 = []
        for i in range(length):
            func = re.sub('t', str(t+h/2.0), functions[i])
            for n in range(length):
                xnr = 'x' + str(n+1)
                func = re.sub(xnr, str(x[n]+a2[n]/2.0), func)
            a3.append(h*eval(func))

        # a4[i]=h*f[i](t+h, x1+a3[1], x2+a3[2], ...)
        a4 = []
        for i in range(length):
            func = re.sub('t', str(t+h), functions[i])
            for n in range(length):
                xnr = 'x' + str(n+1)
                func = re.sub(xnr, str(x[n]+a3[n]), func)
            a4.append(h*eval(func))

        return a1,a2,a3,a4
#----


root = Tkinter.Tk()
root.title('Runge-Kutta visualisation')

if os.name in ('nt', 'posix'):
    root.option_readfile('fontsAndColors.txt')
else:
    Pmw.initialise(root, fontScheme = 'pmw1')

Gui()

root.resizable(0,0)
root.mainloop()
```

Chapter 15

Sorting algorithms

15.1 Introduction

Five different sorting algorithms feature in this section: bucket-sort, insertion-sort, merge-sort, quick-sort, and shell-sort. They are all widely known, well documented and analysed, and may be found in most textbooks covering the subjects of data structures and algorithm analysis (see for example [10] and [11]).

The visual elements used to describe the algorithms are very similar, almost identical in some cases. A lot of time and coding space are therefore saved by collecting common tasks in a separate module and in creating a template file. The individual sorting algorithm files are thus relatively short and contain, in addition to the actual algorithm, canvas positions and sizes, some preference values, and other algorithm specific details.

Two of the algorithms (insertion-sort and shell-sort) use element swapping to sort a sequence. Merge-sort and quick-sort use a divide-and-conquer technique. Bucket-sort uses an altogether different approach.

15.2 Base classes

15.2.1 BaseGui

This class creates the graphical user interface, defines the keyboard shortcuts and button bindings, and handles the event processing. It displays messages and manages object animations.

The program flow will be handled by creating an instance of the `ProgramFlow` class from the `programflow` module (section 9).

All the sorting applications use the same basic GUI: a *Tkinter.Canvas* to visualise the algorithm, three *Pmw.Counters* to set the length of the sequence and the sequence's min/max values, and six *Tkinter.Buttons*. The size of the canvas and the default and min/max values of the counters are set by the individual derived GUI classes (see the template section (15.3)).

The buttons (keyboard shortcuts in parentheses):

- "Redraw sequence" (r) - Create a new sequence of random values. This will also cancel the current algorithm progress.
- "Start animation" (a) - Animate the rest of the algorithm.

- "Step by step" (n) - Execute the next step in the algorithm.
- "Preferences" (p) - Display preference window.
- "Help" (h) - Display reference page.
- "Exit" (q) - Quit application.

The public functions:

- `startup(prefs)` - Start the application by drawing the sequence. `prefs` is a preference object (see the template section (15.3)).
- `swap_elements(element1, element2, type, speed)` - Swap two elements. The elements have to be objects of the `Element` class (see section 15.2.3). `type` can be either "linear" or "curved" and decides how to actually move the elements. `speed` is the speed of the movement. This is an integer value, typically in the range [1, 10] where 1 corresponds to the fastest movement available (see the `programflow` module (section 9) for more details).
- `move_sequence(sequence, dy, speed)` - Move a `Sequence` object (see section 15.2.2) either up ($dy = -1$) or down ($dy = 1$). The actual length of the movement is set by the derived `GUI` class attribute `dy`. This value should be initialised in the `GUI`'s `__init__()` function (see the template section (15.3)). `speed` is the speed of the movement as explained above.
- `move_sequences(sequences, dy, speed)` - This does the same thing as the `move_sequence()` function, but works on a list of sequences.
- `move_element(element, x1, y1, speed)` - Move an `Element` object (section 15.2.3) from its current canvas position to the new position $(x1, y1)$ with the given `speed`. The element's path will be calculated by the `create_path` function found in the `misc` module (section 8).
- `draw_active_pos_arrow(element, tags = 'arrow')` - Draw an arrow with the text "active pos" below the given `element`. The arrow (and the text) will be given the tag `tags`.
- `display_incr_seq(incrSeq, tag = 'incr')` - Display a sequence of numbers (`incrSeq`) in the upper right corner of the canvas. Note that `incrSeq` should be a list, or tuple, of numbers, not a `Sequence` object. The numbers are preceded by the text "Increment:". The text will be given the tag `tag`, and each number a unique tag: `tag` + "_" + a number corresponding to its position in the sequence. This tag may be used to highlight the numbers individually.
- `display_message(text, x = 10, y = 10, anchor = 'nw', pause = 0)` - Display the message `text` at canvas position (x, y). A pause in seconds (`pause`) may be given.
- `display_end_message(y, font = ('Helvetica', 12, 'bold'))` - Display the message "Sort finished!" in the canvas at at centered x-position and at height `y`.

The text font may be explicitly given. This function will also change the appropriate `programflow` parameters (see section 9) to indicate that the algorithm has ended.

```
import Tkinter
import Pmw
import os
import thread
from modules import misc
from modules import programflow
from math import *

class BaseGui:
    """Sets up the graphical user interface, keyboard and button bindings,
       displays messages, handles event processing and object animations"""

    def __init__(self, master):
        self.master = master

        # Use module class to handle algorithm flow
        args = (master, self.sort.start_algorithm)
        kw = {'speedFactor': self.speedFactor}
        self.flow = programflow.ProgramFlow(*args, **kw)

        self.balloon = Pmw.Balloon(self.master, initwait = 500)

        # Create the left counters
        self.seqlenCounter = self._create_counter('Sequence length: ',
                str(self.seqValue[0]),
                str(self.seqValue[1]),
                str(self.seqValue[2]))
        self.minCounter = self._create_counter('Min value:',
                str(self.minValue[0]),
                str(self.minValue[1]),
                str(self.minValue[2]))
        self.maxCounter = self._create_counter('Max value:',
                str(self.maxValue[0]),
                str(self.maxValue[1]),
                str(self.maxValue[2]))

        # Create the middle buttobox
        buttonbox1 = Pmw.ButtonBox(self.master,
                orient = 'vertical',
                pady = 1)
        b1 = buttonbox1.add('Redraw sequence', command = self._redraw_sequence)
        b2 = buttonbox1.add('Start animation', command = self._start_animation)
        b3 = buttonbox1.add('Step by step', command = self.flow.next_step)

        # Create the right buttonbox
        buttonbox2 = Pmw.ButtonBox(self.master,
                orient = 'vertical',
                pady = 1)
        self.prefsBtn = buttonbox2.add('Preferences',
                command = self._launch_prefs_window)
        buttonbox2.add('Help', command = self._help)
        buttonbox2.add('Exit', command = self._exit)
        self.buttons = (b1,b2,b3)

        # Create the canvas
```

```
        self.canvas = Pmw.ScrolledCanvas(self.master,
                canvas_background = 'White',
                hull_height = self.canvasHeight,
                hull_width = self.canvasWidth,
                usehullsize = 1)

        # Position widgets
        self.master.grid_rowconfigure(0, min = 10)
        self.seqlenCounter.grid(column = 0, row = 1, padx = 10)
        self.minCounter.grid(column = 0, row = 2)
        self.maxCounter.grid(column = 0, row = 3)
        buttonbox1.grid(column = 1, row = 1, rowspan = 3, padx = 10)
        self.canvas.grid(row = 4, columnspan = 3, padx = 10, pady = 10)
        buttonbox2.grid(column = 2, row = 1, rowspan = 3, padx = 10)
        Pmw.alignlabels((self.seqlenCounter, self.minCounter, self.maxCounter))

        self._bindings()


#--------------------------
#---- Public functions ----
#--------------------------

    # At startup: draw the sequence
    # Used by: all
    def startup(self, prefs):
        self.prefs = prefs
        self._redraw_sequence()


    # Handle the animated swapping of two elements.
    # Used by: insertion-sort, shell-sort
    def swap_elements(self, element1, element2, type, speed):
        if type == 'linear':
            path = self._calculate_linear_path(element1.x, element2.x)
            factor = 0

        if type == 'curved':
            path = self._calculate_curved_path(element1.x, element2.x)
            factor = 3.5

        for dx,dy in path:
            self.canvas.move(element1.tag, dx, dy)
            self.canvas.move(element2.tag, -dx, -dy)
            self.flow.update_and_pause(speed = speed + factor)

        return element2.x, element1.x


    # Move the given sequence either up (dy=-1) or down (dy=1).
    # Used by: merge-sort, quick-sort
    def move_sequence(self, sequence, dy, speed):
        tags = sequence.get_tags()
        for i in range(self.dy):
            for tag in tags:
                self.canvas.move(tag, 0, dy)
            self.canvas.move(sequence.tag, 0, dy)
            self.canvas.resizescrollregion()
            self.flow.update_and_pause(speed = speed)
```

```
        sequence.set_new_y(dy)


    # Move the given sequences either up (dy=-1) or down (dy=1).
    # Used by: quick-sort
    def move_sequences(self, sequences, dy, speed):
        # Gather tags
        tags = []
        for sequence in sequences:
            tags.extend(sequence.get_tags())
            tags.append(sequence.tag)

        # Move sequences
        for i in range(self.dy):
            for tag in tags:
                self.canvas.move(tag, 0, dy)
            self.flow.update_and_pause(speed = speed)

        # Set new position values
        for sequence in sequences:
            sequence.set_new_y(dy)


    # Move the given element by a straight line to position (x1,y1).
    # Used by: bucket-sort, merge-sort, quick-sort
    def move_element(self, element, x1, y1, speed):
        x0 = element.x
        y0 = element.y
        path = misc.create_path(x0,y0, x1,y1)
        for dx,dy in path:
            self.canvas.move(element.tag, dx, dy)
            self.flow.update_and_pause(speed = speed)


    # Draw an arrow (with text) to mark the active element.
    # Used by: insertion-sort, shell-sort
    def draw_active_pos_arrow(self, element, tags = 'arrow'):
        args = (element.x,element.y+60, element.x, element.y+40)
        kw = {'arrow': 'last', 'arrowshape': (6,8,2), 'tags': tags}
        self.canvas.create_line(*args, **kw)

        kw = {'text': 'active pos', 'tags': tags}
        self.canvas.create_text(element.x, element.y+75, **kw)


    # Display the increment sequence in the upper right corner of the canvas.
    # Give each element in the sequence an uniqe tag so that
    # the active element may be highlighted.
    # Used by: shell-sort
    def display_incr_seq(self, incrSeq, tag = 'incr'):
        n = len(incrSeq)
        kw = {'anchor': 'ne', 'text': 'Increment:', 'tags': tag}
        self.canvas.create_text(self.canvasWidth-10-15*n, 10, **kw)

        for i in range(n):
            tags = (tag, tag + '_' + str(i))
            kw = {'anchor': 'ne', 'text': str(incrSeq[i]), 'tags': tags}
            self.canvas.create_text(self.canvasWidth-15*(n-i), 10, **kw)
```

```
    # Display a message in the canvas (and take a pause)
    # Used by: insertion-sort, merge-sort, shell-sort
    def display_message(self, text, x = 10, y = 10, anchor = 'nw', pause = 0):
        kw = {'text': text, 'anchor': anchor, 'tags': 'message'}
        kw['font'] = ('Helvetica', 8, 'roman')
        self.canvas.create_text(x,y, **kw)
        self.flow.wait_or_next_step(seconds = pause)


    # Display end message and set variables appropriately
    # Used by: all
    def display_end_message(self, y, font = ('Helvetica', 12, 'bold')):
        if os.name == 'nt':
            x = 210
        else:
            x = 270

        kw = {'text': 'Sort finished!', 'fill': 'Red', 'font': font}
        self.canvas.create_text(x,y, **kw)

        self._change_buttons_state('disable', (0,1,1))
        self._change_buttons_state('normal', (1,0,0))

        self.flow.animation = 0
        self.flow.started = 0


#---------------------------
#---- Private functions ----
#---------------------------

    # Create keyboard bindings
    # Used by: all
    def _bindings(self):
        self.master.bind('<r>', self._redraw_sequence)
        self.master.bind('<a>', self._start_animation)
        self.master.bind('<n>', self.flow.next_step)
        self.master.bind('<p>', self._launch_prefs_window)
        self.master.bind('<h>', self._help)
        self.master.bind('<q>', self._exit)


    # Convenience function for the gui setup
    # Used by: all
    def _create_counter(self, label, value, min, max):
        validator = {'validator': 'integer', 'min': min, 'max': max}
        counter = Pmw.Counter(self.master,
                datatype = 'numeric',
                entryfield_validate = validator,
                entryfield_value = value,
                entry_justify = 'center',
                entry_width = 5,
                labelpos = 'w',
                label_text = label)
        return counter


    # Handles the first stages of creating a new sequence
```

```
    # Used by: all
    def _redraw_sequence(self, event = None):
        length = int(self.seqlenCounter.get())
        min = int(self.minCounter.get())
        max = int(self.maxCounter.get())

        if min >= max:
            text = 'Please make sure that the max value\nis greater than '+\
                   'the min value!'
            top = Pmw.MessageDialog(self.master,
                    defaultbutton = 0,
                    message_text = text,
                    title = 'Message')
            misc.process_toplevel(top, self.master, 260, 120)
            return

        self._change_buttons_state('normal')
        self.canvas.delete('all')

        self.flow.started = 0

        # Calculate the left starting point of the sequence
        self.x0 = self.canvasWidth/2 - (length-1)*self.dx/2

        # Let the Sort class handle the actual sequence creation
        args = (self.x0, self.y0, self.dx, self.dy, length, min, max)
        self.sort.redraw_sequence(*args)


    # Set/change the state of the three middle buttons
    # Used by: all
    def _change_buttons_state(self, state, buttons = (1,1,1)):
        if buttons[0]:
            self.buttons[0].configure(state = state)
        if buttons[1]:
            self.buttons[1].configure(state = state)
        if buttons[2]:
            self.buttons[2].configure(state = state)


    # Disable button and animate the rest of the algorithm
    # Used by: all
    def _start_animation(self, event = None):
        self._change_buttons_state('disabled')
        self.flow.start_animation()


    # Launch the preferences window
    # Used by: all
    def _launch_prefs_window(self, event = None):
        self.prefsBtn.configure(state = 'disabled')
        self.prefs.launch_window()


    # Calculate the linear path used in swapping two elements.
    # Used by: insertion-sort, shell-sort
    def _calculate_linear_path(self, x1, x2):
        path = []
        for i in range(25):         # Move up (down)
```

```
            path.append((0,-1))
        for i in range(x1-x2):      # Move left (right)
            path.append((-1,0))
        for i in range(25):         # Move down (up)
            path.append((0,1))
        return path


    # Calculate the curved path used in swapping two elements.
    # Used by: insertion-sort, shell-sort
    def _calculate_curved_path(self, x1, x2):
        path = []
        lastY = 0
        for i in range(1, x1-x2+1):
            y = 30*sin(i*pi/(x1-x2))
            path.append((-1, y-lastY))
            lastY = y
        return path


    # Display the reference page
    # Used by: all
    def _help(self, event = None):
        if os.name == 'nt':
            app = os.path.normpath(self.refPage)
        else:
            app = 'netscape ' + os.path.normpath(self.refPage) + '&'
        thread.start_new_thread(os.system, (app,))


    # Destroy master window
    # Used by: all
    def _exit(self, event = None):
        self.master.destroy()
```

15.2.2 Sequence

Some of the sorting algorithms work on individual elements, but the divide-and-conquer based algorithms (merge-sort and quick-sort) work instead on sequences of elements. The details of the individual elements are then hid in the sequence tasks.

Initialise a `Sequence` object with a list of premaid `Elements` (see section 15.2.3).

Use the following functions to manipulate the sequence:

- `choose_pivot(prefs)` - Choose a pivot element. The method of choice dependes on the `prefs` attribute `pivot`. It should be either "first", "last", or "random".
- `draw(colour = 'Black', font = ('Helvetica', 9, 'roman'))` - Draw all the elements in the sequence. The arguments `colour` and `font` will be forwarded to the individual elements' `draw` functions.
- `highlight(small = 0, colour = 'Black')` - This will highlight the sequence by drawing a `SmoothRectangle` (see the `widgets` module (chapter 10)) around it. The `small` argument may be used to draw a thinner and slightly smaller

rectangle than the default one. The `colour` argument is forwarded to the `SmoothRectangle`'s `ouline` argument.

- `get_tags()` - Collect the tags of the sequence's elements and return it as a list. This function is only used by the `BaseGui` class.
- `set_new_y(dy)` - Update the elements' y-positions. `dy` should be either -1 (up one step) or 1 (down one step). This function is only used by the `BaseGui` class.
- `set_new_elements(elements)` - Replace the elements in the sequence with `elements`.
- `get_first_x()` - Return the x-value of the leftmost element in the sequence.

```
import whrandom
from modules import widgets

class Sequence:
    """Some algorithms use divide and conquer to sort a sequence.
       This class handles common (sub)sequence tasks.
       Used by: merge-sort, quick-sort"""

    def __init__(self, elements):
        self.elements = elements                  # A list of Elements
        self.tag = re.sub('\s', '', str(self))    # Create unique tag


    # Return the pivot element according to the preferred method.
    # Used by: quick-sort
    def choose_pivot(self, prefs):
        if prefs.pivot.lower() == 'first':
            return self.elements[0]
        elif prefs.pivot.lower() == 'last':
            return self.elements[len(self.elements)-1]
        elif prefs.pivot.lower() == 'random':
            return self.elements[whrandom.randint(0,len(self.elements)-1)]


    # Draw the elements in the sequence.
    # Used by: merge-sort, quick-sort
    def draw(self, colour = 'Black', font = ('Helvetica', 9, 'roman')):
        for element in self.elements:
            element.draw(colour, font)


    # Highlight the sequence by drawing a smoothRectangle around it.
    # Two different sizes are available: small and large (default).
    # Used by: merge-sort, quick-sort
    def highlight(self, small = 0, colour = 'Black'):
        if not self.elements:
            return
        canvas = self.elements[0].canvas
        x0 = self.elements[0].x
        y0 = self.elements[0].y
        dx = self.elements[0].dx
        n = len(self.elements)-1

        kw = {'dx': 10, 'dy': 10, 'outline': colour, 'tags': self.tag}
```

```
        if small:
            args = (canvas, x0-9,y0-9, x0+n*dx+9,y0+9)
        else:
            args = (canvas, x0-12,y0-12, x0+n*dx+12,y0+12)
            kw['width'] = 2
        widgets.SmoothRectangle(*args, **kw).draw()


    # Return the tags of all the elements in the sequence.
    # Used by: merge-sort, quick-sort (called only from BaseGui functions)
    def get_tags(self):
        tags = []
        for element in self.elements:
            tags.append(element.tag)
        return tags


    # Update the y value for all elements (after moving the sequence).
    # The dy value is either -1 (up one step) or 1 (down one step)
    # Used by: merge-sort, quick-sort (called only from BaseGui functions)
    def set_new_y(self, dy):
        for element in self.elements:
            element.set_new_y(dy)


    # Replace the elements in the sequence.
    # Used by: merge-sort, quick-sort
    def set_new_elements(self, elements):
        dx = self.elements[0].dx
        x0 = self.get_first_x()
        self.elements = []
        for element in elements:
            element.x = x0
            self.elements.append(element)
            x0 = x0 + dx


    # Return the x value of the leftmost element in the sequence.
    # (It's not necessarily the first element!)
    # Used by: merge-sort, quicks-ort
    def get_first_x(self):
        x = 100000
        for element in self.elements:
            if element.x < x:
                x = element.x
        return x
```

15.2.3 Element

This class handles common individual element tasks. Initialise an object with the following arguments:

- `canvas` - a *Tkinter.Canvas* object.
- `value` - The value of the element. Numbers are well suited for demonstrational purposes, but any type of variable should do.
- `x` - The x-position (in the canvas) of the element.

- `y` - The y-position (in the canvas) of the element.
- `dx = 0` - The distance between elements in a sequence. Used only by the `Sequence` class.
- `dy = 0` - The distance between sequences in the binary tree created by the divide-and-conquer based algorithms. Used only by the `Sequence` class.

The available functions:

- `draw(colour = 'Black', font = ('Helvetica', 9, 'roman'))` - Draw the element with the given colour and font. The tag associated with this element is given by the attribute `tag` which gets an unique value when the object is initialised.
- `highlight(colour = 'Red', font = ('Helvetica', 9, 'roman'))` - Highlight the element by giving it another colour and font.
- `remove_highlight()` - Restore the element's colour and font to their original values.
- `set_new_y(dy)` - Update the element's y-value. This should be done after the element's sequence has been moved. `dy` should be either -1 (up one step) or 1 (down one step). This function is only used by the `Sequence` class.

```
import re

class Element:
    """Handles common individual element tasks"""

    def __init__(self, canvas, value, x, y, dx = 0, dy = 0):
        self.canvas = canvas
        self.value = value
        self.x = x
        self.y = y
        self.dx = dx              # Only used by: merge-sort and quick-sort
        self.dy = dy              # Only used by: merge-sort and quick-sort
        self.colour = 'Black'                     # Used to remove highlight
        self.font = ('Helvetica', 9, 'roman')     # Used to remove highlight
        self.tag = re.sub('\s', '', str(self))    # Create unique tag


    # (Re)draw the element with given colour and font.
    # Used by: bucket-sort, insertion-sort, merge-sort, quick-sort, shell-sort
    def draw(self, colour = 'Black', font = ('Helvetica', 9, 'roman')):
        self.colour = colour
        self.font = font
        self.canvas.delete(self.tag)
        kw = {'text': str(self.value), 'fill': colour, 'font': font,
              'tags': self.tag}
        self.canvas.create_text(self.x, self.y, **kw)


    # Highlight the element with given colour and font.
    # Used by: insertion-sort, quick-sort, shell-sort
    def highlight(self, colour = 'Red', font = ('Helvetica', 9, 'roman')):
        self.canvas.itemconfig(self.tag, fill = colour, font = font)
```

```
    # Return element's normal colour and font.
    # Used by: insertion-sort, quick-sort, shell-sort
    def remove_highlight(self):
        self.canvas.itemconfig(self.tag, fill = self.colour, font = self.font)


    # Update element's y value (after moving sequence)
    # The dy value is either -1 (up one step) or 1 (down one step)
    # Used by: merge-sort, quick-sort (called only from Sequence functions)
    def set_new_y(self, dy):
        self.y = self.y + dy*self.dy
```

15.3 Template

15.3.1 Introduction

This is the basic structure of all the sorting applications. It consists of three classes:

- `Gui` - a subclass of `BaseGui` (see the base classes (section 15.2)).
- `Sort` - handles the specifics of the algorithm.
- `Prefs` - a subclass of `BasePrefs` (see the `dialogs` module (chapter 6)).

These lines of code are necessary to initialise the classes correctly:

```
root = Tkinter.Tk()

gui = Gui()
prefs = Prefs(gui)
gui.startup(prefs)

root.mainloop()
```

See the code (section 15.3.5) for more details.

15.3.2 Gui

Almost all the necessary work is done in the base class. The only function needed is the `__init__` function. The thing to do is to modify the attribute values. This will probably require some testing and should be done after the `Sort` class has been written. Note that not all of the attributes are needed in every application.

The initial `refPage` attribute is a path to the reference page. It is displayed whenever the "help" button, or the "h" key, is pressed.

15.3.3 Sort

This is where most of the work for each application needs to be done. There are three mandatory functions:

- `__init__()` - The initialisation function. Not much needs to be done here. It is usually enough to define either a `None` `Sequence` object or an empty `Element` list. The contents of these attributes should be created in the next function.

- `redraw_sequence(x0, y0, dx, dy, length, min, max)` - This will be called from the `BaseGui` class whenever the "Redraw sequence" button, or the "r" key, is pressed. Even when all the arguments are not needed they are all sent to this function. The values you need later should be defined as class attributes. The main thing to do in this function is to create and draw new elements. In some applications (the divide-and-conquer algorithms for instance) you also need to put the elements in a new sequence. Other visual properties (for example sequence highlighting) may also be applied here.

- `start_algorithm()` - This function is called whenever the demonstration is started by eiher pressing the "Start animation" button or the "Step by step" button for the first time. The algorithm should be started by calling a function that performes the necessary steps in a sequential manner.

As mentioned above we now have to write the function that handles the algorithm. In this function we probably need access to the functions in the `BaseGui` and `ProgramFlow` classes. If the application is initialised as we did in the introduction, these should be available by preceeding their names with "gui." and "gui.flow.", respectively. Preference values, like `speed` and `pause`, should be class attributes of the `Prefs` class discussed below. The attributes are available by preceeding their names with "prefs.". For example:

```
gui.move_sequence(sequence, 1, prefs.speed)
gui.flow.wait_or_next_step(seconds = prefs.pause)
```

15.3.4 Prefs

This is a subclass of the `BasePrefs` class. Use it to define preference values that may be changed by the user of the application. The `dialogs` module (chapter 6) contain documentation on how to use it.

15.3.5 Code

```
#!/bin/sh
""":"
exec python $0 ${1+"$@"}
"""

# Make sure the modules are found
import sys
sys.path[:0] = ['../']

# This will enable the program to be run from idle
import os
dirname,basename = os.path.split(sys.argv[0])
os.chdir(os.path.normpath(dirname))

import Tkinter
import Pmw
```

```
import whrandom
from sorting import baseclasses
from modules import dialogs


class Gui(baseclasses.BaseGui):
    def __init__(self):
        # Path to the reference page
        self.refPage = '../doc/' + 'applications_sorting_****.html'

        self.x0 = 50                # Start position of sequence (x-coordinate)
        self.y0 = 50                # Start position of sequence (y-coordinate)
        self.dy = 50                # Might be individual...
        self.seqValue = (10, 2, 14) # Sequence length (default, min, max)
        self.minValue = (1, -9, 8)  # Element min value (default, min, max)
        self.maxValue = (9, -8, 9)  # Element max value (default, min, max)
        self.canvasHeight = 380

        if os.name == 'nt':
            self.dx = 22            # Distance between elements in sequence
            self.canvasWidth = 420
            self.speedFactor = 6000
        else:
            self.dx = 30            # Distance between elements in sequence
            self.canvasWidth = 540
            self.speedFactor = 2000

        # Initialise class to handle algorithm
        self.sort = Sort()

        # Initialise base class
        baseclasses.BaseGui.__init__(self, root)


class Sort:
    def __init__(self):
        pass
#        self.sequence = None
#        self.elements = []

    def redraw_sequence(self, x0, y0, dx, dy, length, min, max):
        pass

    def start_algorithm(self):
        pass


class Prefs(dialogs.BasePrefs):
    def __init__(self, gui):
        dialogs.BasePrefs.__init__(self, gui.balloon)

        # Create the preference values as class attributes here

    def launch_window(self):
        if os.name == 'nt':
            height = 390
        else:
            height = 420
```

```
        self.create_dialog(root, width = 240, height = height)

        # All counters may be specified as a list of this type:
        # (label, variable, type, min, max, balloon_text)
        list = ()
        self.counters = self.create_counters(list)

        # Other widgets types may also be create

    def close_window(self, button):
        if button == 'OK':
            # Reset the attribute values
            pass
        self.dialog.destroy()
        gui.prefsBtn.configure(state = 'normal')


root = Tkinter.Tk()
root.title('****-sort visualization')

if os.name in ('nt', 'posix'):
    root.option_readfile('fontsAndColors.txt')
else:
    Pmw.initialise(root, fontScheme = 'pmw1')

gui = Gui()
prefs = Prefs(gui)
gui.startup(prefs)

root.resizable(0,0)
root.mainloop()
```

15.4 Bucket-sort

15.4.1 About bucket-sort

Bucket-sort is an algorithm with some restrictions on the type of elements allowed. The number of different values has to be finite, as each element value have to correspond to an individual bucket. The bucket is usually represented by some sort of an array. If for example all the elements are non-negative integers (with a reasonably low maximum value), each element with value i increments the value in array[i] by one. The array is initialised by zeros. Go through the array and print out the array index as many times as the value of each array element to get the sorted sequence.

More general sequence elements may be used. Just define a class `Bucket` with an attribute value that corresponds to the element type. As a `Bucket` object has to be created for each possible element value there still has to be a managable number of different values allowed. Keep the `Bucket` objects sorted, with respect to the attribute value, in an array and proceed as above.

A more thorough investigation of bucket-sort's properties can be found in [10] and [11].

15.4.2 The algorithm

- Create an array with elements that correspond to each of the possible sequence element values. This may involve defining a class `Bucket` with a value attribute and filling the array with `Bucket` objects. In any case I will use the term bucket as the element container henceforth.
- For each element in the unsorted sequence: Find the corresponding bucket. Increment the bucket's size or append the element to the bucket's element list.
- For each bucket (in sorted order): Print the bucket's value attribute as many times as the size of the bucket (or the length of the bucket's element list) dictates.

15.4.3 The illustration

To keep the number of buckets managable, the minimum and maximum value of the sequence elements are set to 0 and 9, thus limiting the maximum number of buckets to 10. If the "Redraw sequence" button is pressed, create the appropriate buckets according to the "Min value" and "Max value" counters. The buckets are drawn with their element value printed below (see figures 15.1 - 15.3). When the algorithm starts, move each element to its corresponding bucket. Take a pause (in animation mode) or halt the algorithm (in step by step mode) after each element has been moved to its bucket. Start to empty the buckets when there are no elements left. Start with the left bucket (the bucket with the lowest value). For each element removed from the bucket, take a pause (in animation mode) or halt the algorithm (in step by step mode).

The pauses and the element movement speed may be adjusted in the preferences window.

15.4.4 Code with comments

The `Bucket` class has been created to make the algorithm as general as possible. Each `Bucket` object has an attribute `n` that corresponds to a possible sequence element value, a list of the elements put in the bucket, and some attribute values to keep tab of where (in the canvas) to move the elements.

The `Sort` class has a list of the bucket objects created in sorted order at the same time as the elements are created (in the `redraw_sequence` function).

The rest of the code should be a straightforward rendering of the algorithm.

```
#!/bin/sh
""":"
exec python $0 ${1+"$@"}
"""

# Make sure the modules are found
import sys
sys.path[:0] = ['../']

# This will enable the program to be run from idle
import os
dirname,basename = os.path.split(sys.argv[0])
os.chdir(os.path.normpath(dirname))

import Tkinter
```

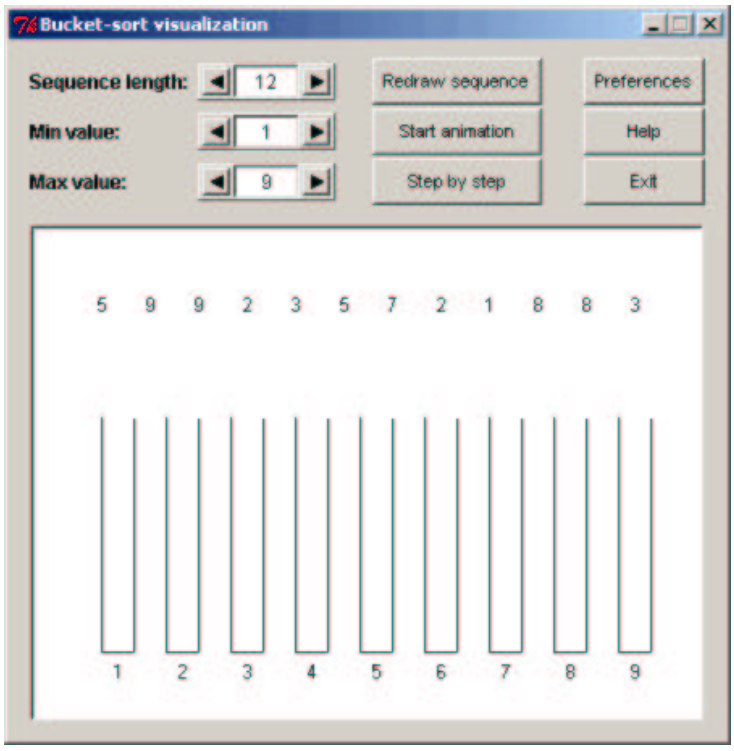


Figure 15.1: Before starting the algorithm.

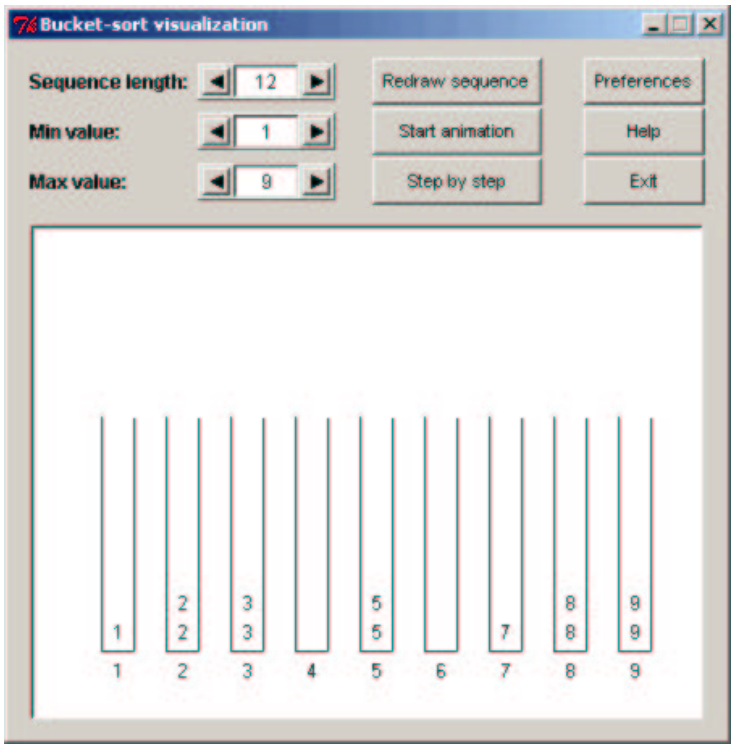


Figure 15.2: All elements are placed in the buckets.

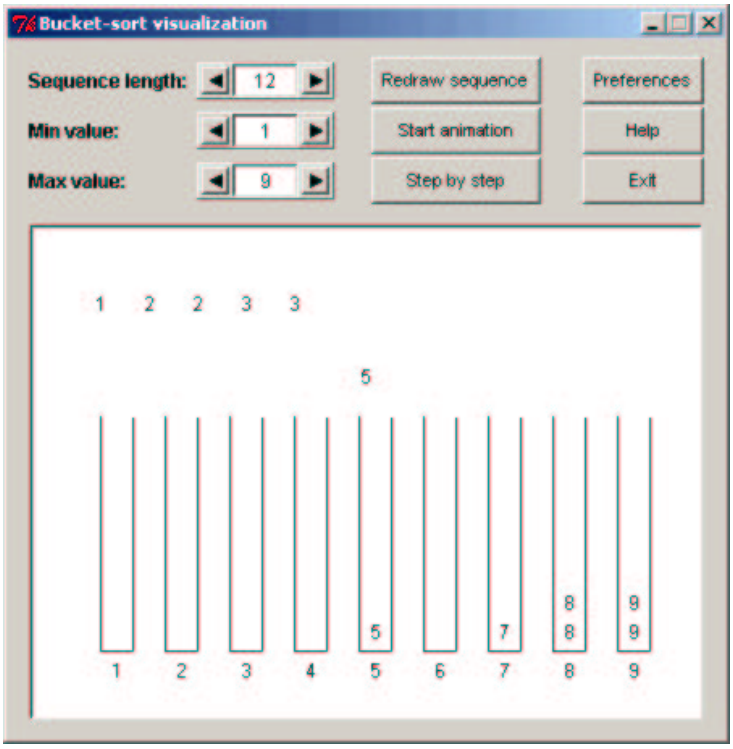


Figure 15.3: Emptying the buckets from left to right.

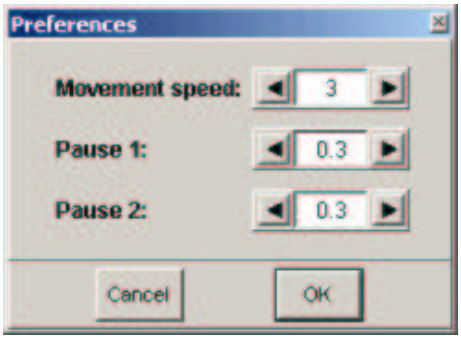


Figure 15.4: The preference window.

```
import Pmw
import whrandom
from sorting import baseclasses
from modules import dialogs

class Gui(baseclasses.BaseGui):
    def __init__(self):
        # Path to the reference page
        self.refPage = '../doc/' + 'applications_sorting_bucket.html'

        self.x0 = 50               # Start position of sequence (x-coordinate)
        self.y0 = 50               # Start position of sequence (y-coordinate)
        self.dy = 20               # Not actaully needed for bucketsort
        self.seqValue = (8, 2, 12) # Sequence length (default, min, max)
        self.minValue = (0, 0, 8)  # Element min value (default, min, max)
        self.maxValue = (9, 1, 9)  # Element max value (default, min, max)
        self.canvasHeight = 310

        if os.name == 'nt':
            self.dx = 30           # Distance between elements in sequence
            self.canvasWidth = 420
            self.speedFactor = 3000
        else:
            self.dx = 40           # Distance between elements in sequence
            self.canvasWidth = 540
            self.speedFactor = 1500

        # Initialise class to handle algorithm
        self.sort = Sort()

        # Initialise base class
        baseclasses.BaseGui.__init__(self, root)
#----

class Bucket:
    def __init__(self, x0, y0, dx, dy, n, canvas):
        self.n = n
        self.elements = []

        self.x = x0 + dx/2
        self.y0 = y0 + dy - 12
        self.yTop = y0 - 5

        canvas.create_line(x0,y0, x0,y0+dy, x0+dx,y0+dy, x0+dx,y0)
        canvas.create_text(x0+dx/2,y0+dy+12, text = str(n))

    def add_element(self, element):
        self.elements.append(element)
        self.y0 = self.y0 - 18
#----

class Sort:
    def __init__(self):
        self.elements = []
        self.buckets = []

    def redraw_sequence(self, x0, y0, dx, dy, length, min, max):
        self.x0 = x0
        self.y0 = y0
```

```
        self.dx = dx
        self.elements = []

        # Create elements
        for i in range(length):
            value = whrandom.randint(min, max)
            element = baseclasses.Element(gui.canvas, value, x0+i*dx, y0)
            element.draw()
            self.elements.append(element)

        if os.name == 'nt':
            x = 205
            dx = 40
        else:
            x = 260
            dx = 50
        x0 = x - (max-min)*dx/2

        # Create buckets
        self.buckets = []
        for i in range(max-min+1):
            bucket = Bucket(x0+i*dx, 120, dy, 145, min+i, gui.canvas)
            self.buckets.append(bucket)

    def start_algorithm(self):
        self.bucketsort()

    def bucketsort(self):
        for element in self.elements:

            # Find the correct bucket
            for bucket in self.buckets:
                if bucket.n == element.value:
                    break

            x1 = bucket.x
            y1 = bucket.yTop
            x2 = bucket.x
            y2 = bucket.y0

            # Move element to the top of the bucket
            gui.move_element(element, x1, y1, prefs.speed)
            element.x = x1
            element.y = y1

            # Move element to the first available position in the bucket
            gui.move_element(element, x2, y2, prefs.speed)
            element.x = x2
            element.y = y2

            # Add element to bucket and pause
            bucket.add_element(element)
            gui.flow.wait_or_next_step(seconds = prefs.pause1)

        n = 0
        for bucket in self.buckets:
            bucket.elements.reverse()
            for element in bucket.elements:
                x1 = bucket.x
```

```
                y1 = bucket.yTop
                x2 = self.x0+n*self.dx
                y2 = self.y0

                # Move element to the top of the bucket
                gui.move_element(element, x1, y1, prefs.speed)
                element.x = x1
                element.y = y1

                # Move element to the first available position in the sequence
                gui.move_element(element, x2, y2, prefs.speed)
                element.x = x2
                element.y = y2

                n = n + 1
                if not n == len(self.elements):
                    gui.flow.wait_or_next_step(seconds = prefs.pause2)

        gui.display_end_message(87)
#----


class Prefs(dialogs.BasePrefs):
    def __init__(self, gui):
        dialogs.BasePrefs.__init__(self, gui.balloon)

        self.speed = 3    # Element movement speed
        self.pause1 = 0.3 # Pause after moving an element to a bucket
        self.pause2 = 0.3 # Pause after moving an element to its sorted place

    def launch_window(self):
        if os.name == 'nt':
            height = 157
        else:
            height = 180

        self.create_dialog(root, width = 240, height = height)
        Tkinter.Frame(self.dialog.interior()).pack(pady = 4)

        list = (('Movement speed: ', self.speed, 'integer', 1, 10,
                 'Element movement speed (1:fast, 10:slow)'),
                ('Pause 1: ', self.pause1, 'real', 0.0, 10.0,
                 'Pause (in seconds) after moving an element to a bucket'),
                ('Pause 2: ', self.pause2, 'real', 0.0, 10.0,
                 'Pause (in seconds) after moving an element to its sorted '+\
                 'place'))
        self.counters = self.create_counters(list)

    def close_window(self, button):
        if button == 'OK':
            self.speed = int(self.counters[0].get())
            self.pause1 = float(self.counters[1].get())
            self.pause2 = float(self.counters[2].get())
        self.dialog.destroy()
        gui.prefsBtn.configure(state = 'normal')
#----

root = Tkinter.Tk()
root.title('Bucket-sort visualization')
```

```
if os.name in ('nt', 'posix'):
    root.option_readfile('fontsAndColors.txt')
else:
    Pmw.initialise(root, fontScheme = 'pmw1')

gui = Gui()
prefs = Prefs(gui)
gui.startup(prefs)

root.resizable(0,0)
root.mainloop()
```

15.5 Insertion-sort

15.5.1 About insertion-sort

This is one of the simplest sorting algorithms available. In general it is not very efficient, but will run quickly if the sequence is almost sorted.

More information on insertion-sort may be found in [11].

15.5.2 The algorithm

Let n be the length of the sequence. For each element in position 2 through n: Let p be the position of the element. Move the element left until its correct place is found among the first p elements, which are now in sorted order.

15.5.3 The illustration

The active element is highlighted in red. An arrow with the text "active pos" points to its position from below. When comparing the element with its left side neighbour, the text "Comparing values" is shown in the upper left corner of the canvas, and the left element is highlighted in purple. There are two ways to move the elements while changing their positions. A curved and a linear movement. Change the type (and the speed) in the preference window. The algorithm takes a pause or halts the execution (animation mode or step by step mode) when:

- A new active element is highlighted (`pause1`).
- The "Comparing values" text is shown (`pause2`).
- An element has finished swapping positions (`pause3`).

The length of the pauses may be set individually in the preferences window.

15.5.4 Code

```
#!/bin/sh
""":"
exec python $0 ${1+"$@"}
"""

# Make sure the modules are found
import sys
```

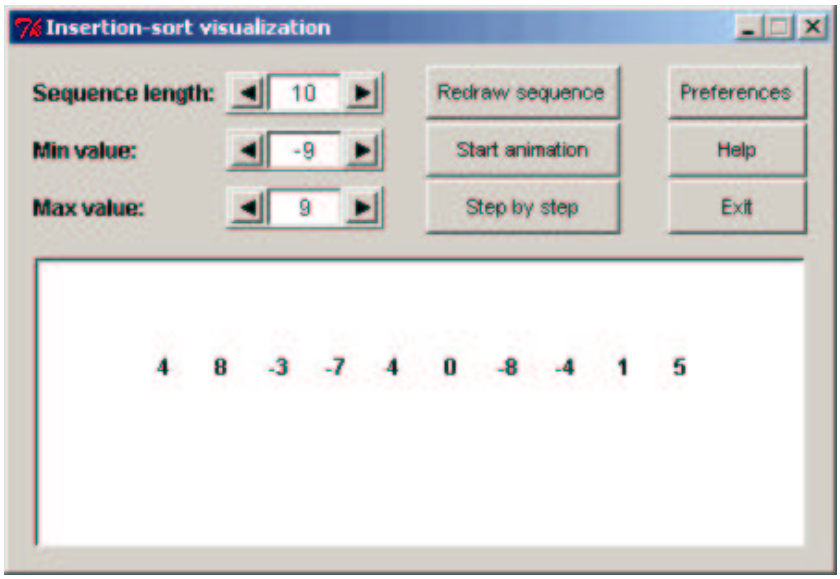


Figure 15.5: Before starting the algorithm.

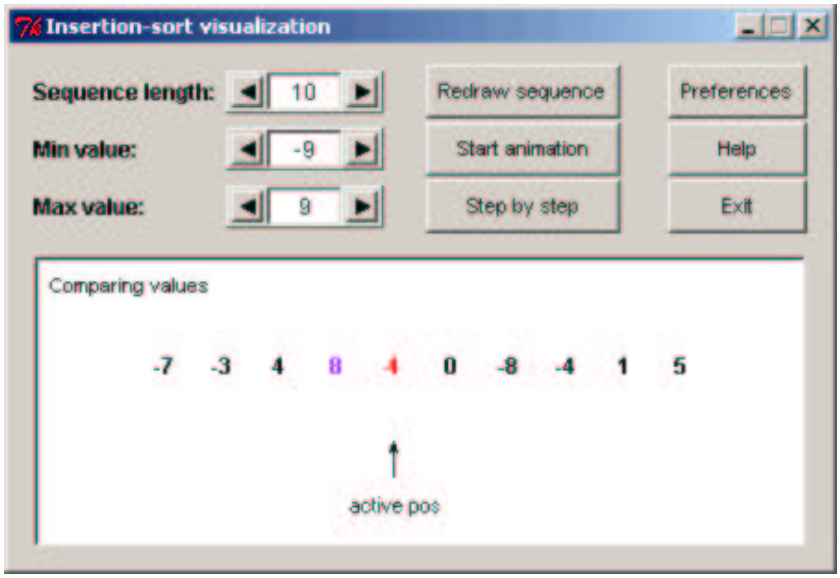


Figure 15.6: Comparing two elements.

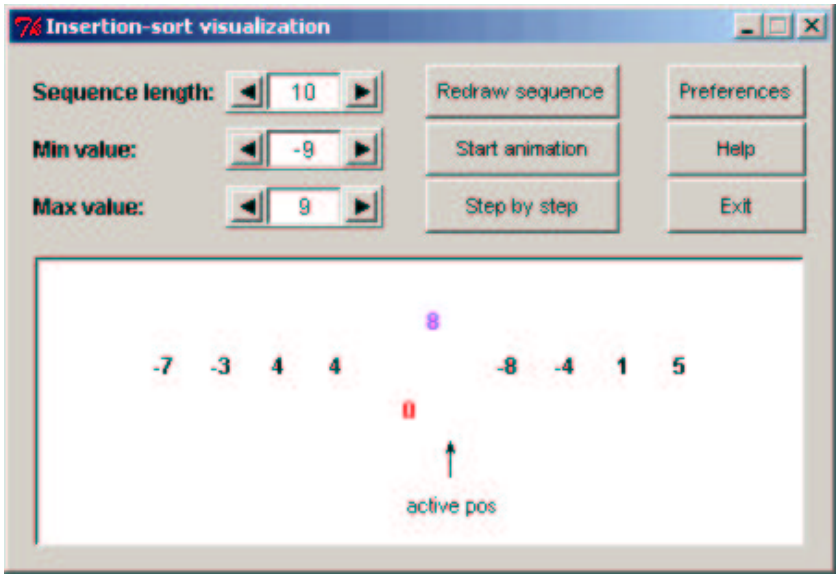


Figure 15.7: Swapping two elements' positions.

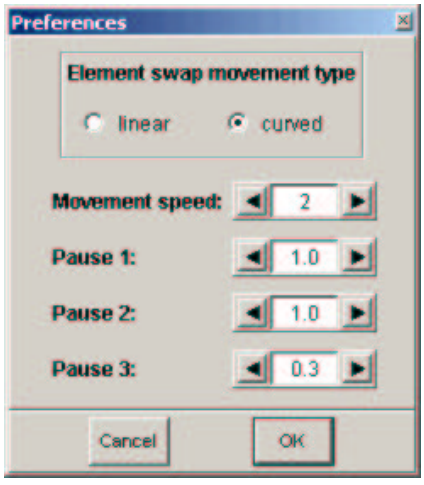


Figure 15.8: The preference window.

```
sys.path[:0] = ['../']

# This will enable the program to be run from idle
import os
dirname,basename = os.path.split(sys.argv[0])
os.chdir(os.path.normpath(dirname))

import Tkinter
import Pmw
import whrandom
from sorting import baseclasses
from modules import dialogs


class Gui(baseclasses.BaseGui):
    def __init__(self):
        # Path to the reference page
        self.refPage = '../doc/' + 'applications_sorting_insertion.html'

        self.x0 = 50                # Start position of sequence (x-coordinate)
        self.y0 = 60                # Start position of sequence (y-coordinate)
        self.dy = 40                # Height when swapping elements
        self.seqValue = (10, 3, 12) # Sequence length (default, min, max)
        self.minValue = (1, -9, 8)  # Element min value (default, min, max)
        self.maxValue = (9, -8, 9)  # Element max value (default, min, max)
        self.canvasHeight = 160

        if os.name == 'nt':
            self.dx = 31            # Distance between elements in sequence
            self.canvasWidth = 420
            self.speedFactor = 8000
        else:
            self.dx = 40            # Distance between elements in sequence
            self.canvasWidth = 540
            self.speedFactor = 3000

        # Initialise class to handle algorithm
        self.sort = Sort()

        # Initialise base class
        baseclasses.BaseGui.__init__(self, root)
#----

class Sort:
    def __init__(self):
        self.elements = []

    def redraw_sequence(self, x0, y0, dx, dy, length, min, max):
        self.elements = []
        for i in range(length):
            value = whrandom.randint(min, max)
            element = baseclasses.Element(gui.canvas, value, x0+i*dx, y0)
            element.draw(font = ('Helvetica', 9, 'bold'))
            self.elements.append(element)

    def start_algorithm(self):
        self.insertionsort()

    def insertionsort(self):
```

```
        for element in self.elements[1:]:
            gui.draw_active_pos_arrow(element)
            element.highlight(font = ('Helvetica', 9, 'bold'))
            gui.flow.wait_or_next_step(seconds = prefs.pause1)

            index = self.elements.index(element)
            while index:
                element2 = self.elements[index-1]
                kw = {'colour': 'Purple', 'font': ('Helvetica', 9, 'bold')}
                element2.highlight(**kw)

                gui.display_message('Comparing values', pause = prefs.pause2)
                gui.canvas.delete('message')

                if element.value < element2.value:
                    args = (element, element2, prefs.swapType, prefs.speed)
                    element.x,element2.x = gui.swap_elements(*args)

                    self.elements[index] = element2
                    self.elements[index-1] = element
                    index = index - 1
                else:
                    index = 0

                element2.remove_highlight()

            element.remove_highlight()
            gui.canvas.delete('arrow')
            gui.flow.wait_or_next_step(seconds = prefs.pause3)

        gui.display_end_message(30, font = ('Helvetica', 10, 'bold'))
#----


class Prefs(dialogs.BasePrefs):
    def __init__(self, gui):
        dialogs.BasePrefs.__init__(self, gui.balloon)

        self.swapType = 'curved' # Swap movement type ('linear','curved')
        self.speed = 3           # Element movement speed
        self.pause1 = 1.0        # Pause after highlighting active element
        self.pause2 = 1.0        # Pause when comparing elements
        self.pause3 = 0.3        # Pause when an element has finished swapping

    def launch_window(self):
        if os.name == 'nt':
            height = 258
        else:
            height = 280

        self.create_dialog(root, width = 240, height = height)

        args = ('Element swap movement type', ('linear', 'curved'), 8)
        self.radioSelect = self.create_radioselect(*args)
        self.radioSelect.invoke(self.swapType)

        list = (('Movement speed: ', self.speed, 'indeger', 1, 10,
                 'Element movement speed (1:fast, 10:slow)'),
                ('Pause 1: ', self.pause1, 'real', 0.0, 10.0,
```

```
                'Pause (in seconds) after highlighting active element'),
               ('Pause 2: ', self.pause2, 'real', 0.0, 10.0,
                'Pause (in seconds) when comparing elements'),
               ('Pause 3: ', self.pause3, 'real', 0.0, 10.0,
                'Pause (in seconds) when an element has finished swapping'))
        self.counters = self.create_counters(list)

    def close_window(self, button):
        if button == 'OK':
            self.swapType = self.radioSelect.getcurselection()
            self.speed = int(self.counters[0].get())
            self.pause1 = float(self.counters[1].get())
            self.pause2 = float(self.counters[2].get())
            self.pause3 = float(self.counters[3].get())
        self.dialog.destroy()
        gui.prefsBtn.configure(state = 'normal')
#----

root = Tkinter.Tk()
root.title('Insertion-sort visualization')

if os.name in ('nt', 'posix'):
    root.option_readfile('fontsAndColors.txt')
else:
    Pmw.initialise(root, fontScheme = 'pmw1')

gui = Gui()
prefs = Prefs(gui)
gui.startup(prefs)

root.resizable(0,0)
root.mainloop()
```

15.6 Merge-sort

15.6.1 About merge-sort

Merge-sort is a recursive algorithm. It uses the general divide-and-conquer technique. The algorithm is found in both [10] and [11], but with quite different visual approaches. This application is largely based on the visual representation found in [10].

15.6.2 The algorithm

Let's assume that we have a sequence s with n elements.

- If s has two or more elements, divide the sequence into two new sequences, s_1 and s_2, so that s_1 contain the first $\lfloor n/2 \rfloor$ elements of s, and s_2 contain the remaining $\lceil n/2 \rceil$ elements.
- Sort the sequences s_1 and s_2 recursively.
- Merge the sorted sequences s_1 and s_2 into a single sorted sequence s.

The divide step of the algorithm should be straightforward, but the merge step requires an explanation. Remove the smallest element from either s_1 and s_2 and add it to the end of the sequence s. Continue this step until both s_1 and s_2 are empty.

15.6.3 The illustration

A `SmoothRectangle` (see the `widgets` module (chapter 10)) with a thick border is used to surround the sequence s. Two smaller and thinner `SmoothRectangles` surround the subsequences s_1 and s_2. Move the subsequences down one step when they are sorted. A "Merging:" text is put below two sequences in the merge step. The elements are then moved one at a time to form the new sorted sequence, which is moved up to the next level. This process is repeated until the algorithm has finished.

The movement speed of the sequences (`speed1`) and the movement speed of the elements (`speed2`) may be adjusted separately in the preference window. The several different pauses (in animation mode; the algorithm halts in step by step mode) may also be set individually. They occur:

- When a sequence is divided (`pause1`).
- When a subsequence has finished its recursive sort (`pause2`).
- Just before two sequences merge (`pause3`).
- Just after two sequences have merged (`pause4`).
- Before a sequence moves up one step (`pause5`).
- Between each element in the merge process (`pause6`).

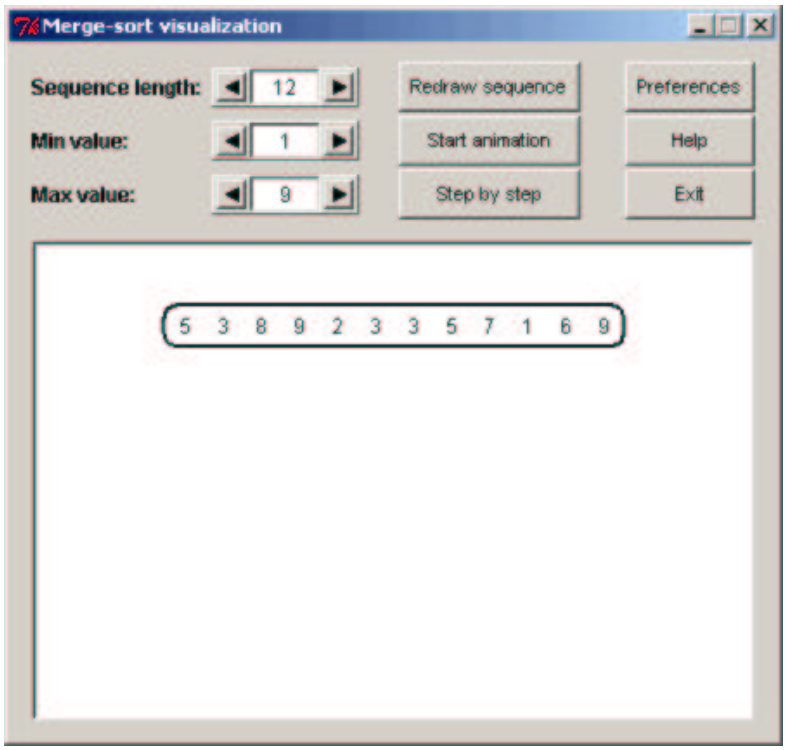


Figure 15.9: Before starting the algorithm.

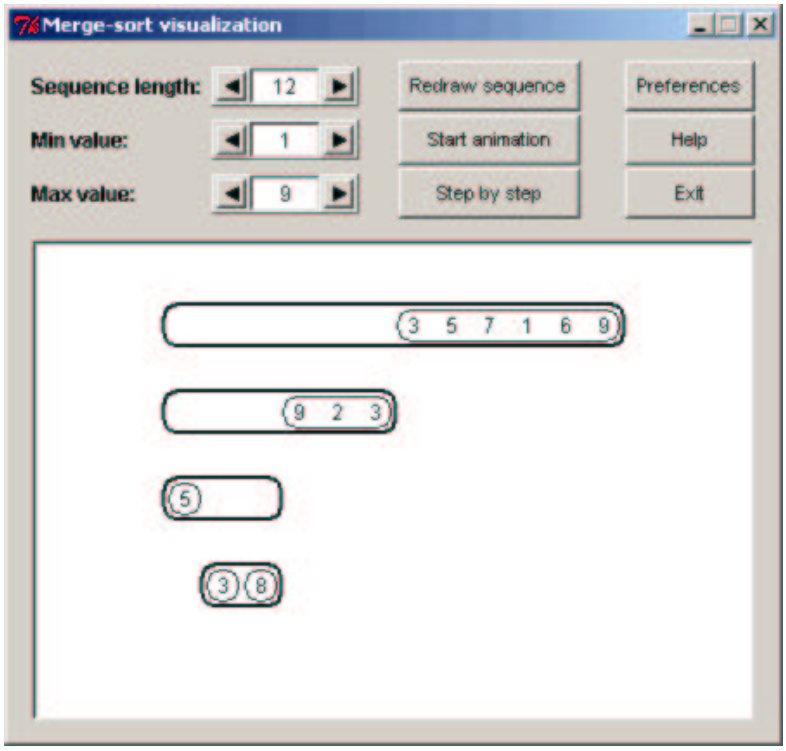


Figure 15.10: The tree structure of the divide-and-conquer technique is clearly visible.

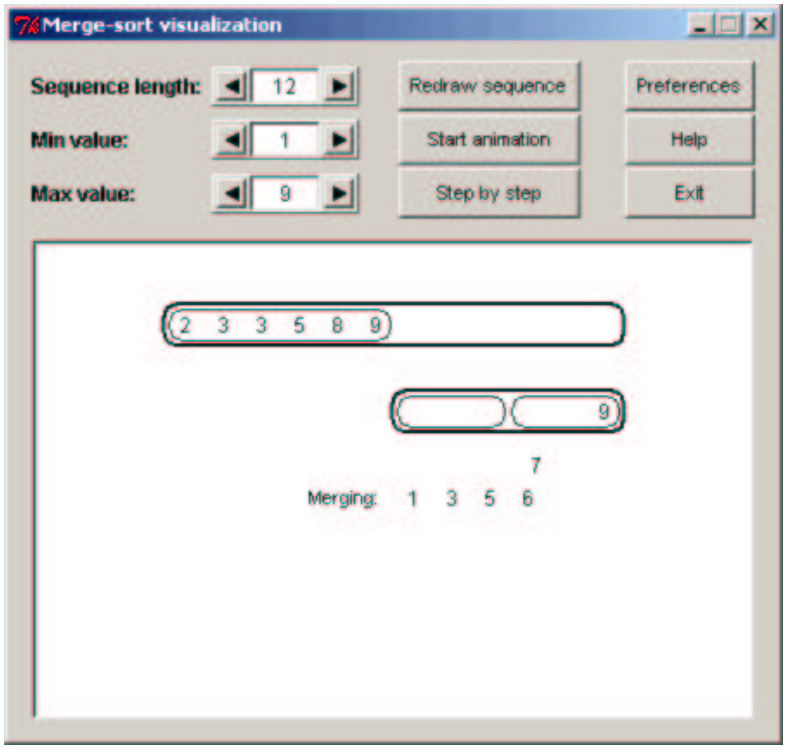


Figure 15.11: Two sequences merge to one.

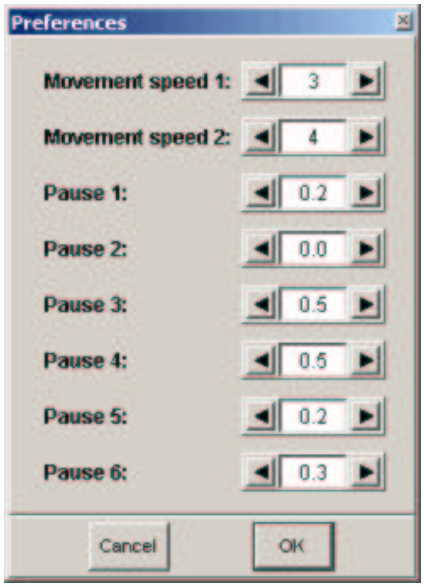


Figure 15.12: The preference window.

15.6.4 Code

```
#!/bin/sh
""":"
exec python $0 ${1+"$@"}
"""

# Make sure the modules are found
import sys
sys.path[:0] = ['../']

# This will enable the program to be run from idle
import os
dirname,basename = os.path.split(sys.argv[0])
os.chdir(os.path.normpath(dirname))

import Tkinter
import Pmw
import whrandom
from sorting import baseclasses
from modules import dialogs


class Gui(baseclasses.BaseGui):
    def __init__(self):
        # Path to the reference page
        self.refPage = '../doc/' + 'applications_sorting_merge.html'

        self.x0 = 50               # Start position of sequence (x-coordinate)
        self.y0 = 50               # Start position of sequence (y-coordinate)
        self.dy = 50               # Distance between each sequence level
        self.seqValue = (8, 2, 14) # Sequence length (default, min, max)
        self.minValue = (1, -9, 8) # Element min value (default, min, max)
        self.maxValue = (9, -8, 9) # Element max value (default, min, max)
```

```
        self.canvasHeight = 280

        if os.name == 'nt':
            self.dx = 22           # Distance between elements in sequence
            self.canvasWidth = 420
            self.speedFactor = 10000
        else:
            self.dx = 29           # Distance between elements in sequence
            self.canvasWidth = 540
            self.speedFactor = 2000

        # Initialise class to handle algorithm
        self.sort = Sort()

        # Initialise base class
        baseclasses.BaseGui.__init__(self, root)
#----

class Sort:
    def __init__(self):
        self.sequence = None

    def redraw_sequence(self, x0, y0, dx, dy, length, min, max):
        self.y0 = y0
        self.dx = dx
        self.dy = dy

        elements = []
        for i in range(length):
            value = whrandom.randint(min, max)
            args = (gui.canvas, value, x0+i*dx, y0, dx, dy)
            element = baseclasses.Element(*args)
            elements.append(element)

        self.sequence = baseclasses.Sequence(elements)
        self.sequence.draw()
        self.sequence.highlight()

    def start_algorithm(self):
        self.mergesort(self.sequence)

    def mergesort(self, sequence):
        if len(sequence.elements) > 1:
            sequence.highlight()
            elements1,elements2 = self.divide(sequence.elements)
            sequence1 = baseclasses.Sequence(elements1)
            sequence1.highlight(small = 1)
            sequence2 = baseclasses.Sequence(elements2)
            sequence2.highlight(small = 1)
            gui.flow.wait_or_next_step(seconds = prefs.pause1)  # Pause 1

            gui.move_sequence(sequence1, 1, prefs.speed1)
            self.mergesort(sequence1)

            gui.flow.wait_or_next_step(seconds = prefs.pause2)   # Pause 2

            gui.move_sequence(sequence2, 1, prefs.speed1)
            self.mergesort(sequence2)
```

```
            x = sequence1.get_first_x()
            y = sequence1.elements[0].y
            kw = {'x': x-40, 'y': y+self.dy, 'anchor': 'center',
                  'pause': prefs.pause3}
            gui.display_message('Merging:', **kw)

            args = (sequence1.elements,sequence2.elements, x, y+self.dy)
            elements = self.merge(*args)

            gui.canvas.delete(sequence1.tag, sequence2.tag)
            sequence.set_new_elements(elements)
            sequence.draw()
            sequence.highlight(small = 1)

            gui.flow.wait_or_next_step(seconds = prefs.pause4)  # Pause 4

            gui.canvas.delete('message')
            gui.move_sequence(sequence, -1, prefs.speed1)

        if not sequence.elements[0].y == self.y0:
            gui.flow.update_and_pause(seconds = prefs.pause5)   # Pause 5
            gui.move_sequence(sequence, -1, prefs.speed1)
        else:
            sequence.highlight()
            gui.display_end_message(135)

    def divide(self, elements):
        n = len(elements)
        elements1 = elements[:n/2]
        elements2 = elements[n/2:]
        return elements1,elements2

    def merge(self, elements1, elements2, x, y):
        elements = []
        while (elements1 and elements2):
            if elements1[0].value <= elements2[0].value:
                element = elements1.pop(0)
            else:
                element = elements2.pop(0)
            elements,x = self.append_and_move_element(element, elements, x, y)

        for element in elements1:
            elements,x = self.append_and_move_element(element, elements, x, y)

        for element in elements2:
            elements,x = self.append_and_move_element(element, elements, x, y)

        return elements

    def append_and_move_element(self, element, elements, x, y):
        gui.flow.update_and_pause(seconds = prefs.pause6)   # Pause 6
        elements.append(element)
        gui.move_element(element, x, y, prefs.speed2)
        element.x = x
        element.y = y
        x = x + self.dx
        return elements,x
#----
```

```
class Prefs(dialogs.BasePrefs):
    def __init__(self, gui):
        dialogs.BasePrefs.__init__(self, gui.balloon)

        self.speed1 = 4     # Sequence movement speed (1:fast, 10:slow)
        self.speed2 = 3     # Element movement speed (during merging)
        self.pause1 = 0.2   # Pause after dividing
        self.pause2 = 0.0   # Pause after first sequence has ended
        self.pause3 = 0.5   # Pause before merging
        self.pause4 = 0.5   # Pause after merging
        self.pause5 = 0.2   # Pause when moving sequence back
        self.pause6 = 0.3   # Pause between each element when merging

    def launch_window(self):
        if os.name == 'nt':
            width = 240
            height = 320
        else:
            width = 260
            height = 360

        self.create_dialog(root, width = width, height = height)
        Tkinter.Frame(self.dialog.interior()).pack(pady = 4)

        list = (('Movement speed 1: ', self.speed1, 'integer', 1, 10,
                 'Sequence movement speed (1:fast, 10:slow)'),
                ('Movement speed 2: ', self.speed2, 'integer', 1, 10,
                 'Element movement speed (during merging) (1:fast, 10:slow)'),
                ('Pause 1: ', self.pause1, 'real', 0.0, 10.0,
                 'Pause (in seconds) after dividing a sequence'),
                ('Pause 2: ', self.pause2, 'real', 0.0, 10.0,
                 'Pause (in seconds) after first sequence has ended'),
                ('Pause 3: ', self.pause3, 'real', 0.0, 10.0,
                 'Pause (in seconds) before merging'),
                ('Pause 4: ', self.pause4, 'real', 0.0, 10.0,
                 'Pause (in seconds) after merging'),
                ('Pause 5: ', self.pause5, 'real', 0.0, 10.0,
                 'Pause (in seconds) when moving a sequence back'),
                ('Pause 6: ', self.pause6, 'real', 0.0, 10.0,
                 'Pause (in seconds) between each element when merging'))
        self.counters = self.create_counters(list)

    def close_window(self, button):
        if button == 'OK':
            self.speed1 = int(self.counters[0].get())
            self.speed2 = int(self.counters[1].get())
            self.pause1 = float(self.counters[2].get())
            self.pause2 = float(self.counters[3].get())
            self.pause3 = float(self.counters[4].get())
            self.pause4 = float(self.counters[5].get())
            self.pause5 = float(self.counters[6].get())
            self.pause6 = float(self.counters[7].get())
        self.dialog.destroy()
        gui.prefsBtn.configure(state = 'normal')
#----

root = Tkinter.Tk()
root.title('Merge-sort visualization')
```

```
if  os.name in ('nt', 'posix'):
    root.option_readfile('fontsAndColors.txt')
else:
    Pmw.initialise(root, fontScheme = 'pmw1')

gui = Gui()
prefs = Prefs(gui)
gui.startup(prefs)

root.resizable(0,0)
root.mainloop()
```

15.7 Quick-sort

15.7.1 About quick-sort

Like merge-sort, quick-sort is a recursive algorithm based on the divide-and-conquer technique. However, the bulk of the work in quick-sort is done at the dividing step, not in the conquer step. The running time of quick-sort is largely dependant on the pivot strategy. Three different choices are here implemented.

The algorithm is described in both [10] and [11], and like merge-sort, this application is largely based on the visual representation found in [10].

15.7.2 The algorithm

We start the algorithm with a sequence s.

- If s has two or more elements, choose an element as the pivot. The three implemented choices for the pivot are: the first element, the last element, and a random element. Create three new sequences s_L, s_E, and s_G, containing the elements in s that are less than the pivot, equal to the pivot, and greater than the pivot, respectively.
- Sort the sequences s_L and s_G recursively.
- Recreate the sequence s by first inserting the elements from s_L, then the elements from s_E, and finally the elements from s_G.

15.7.3 The illustration

A `SmoothRectangle` (see the widgets module (chapter 10)) with a thick border is used to surround the sequence s. The sorting procedure starts by picking the pivot and highlighting it with red colour. The subsequences s_L, s_E and s_G are created by first moving the smaller elements, the equal elements and finally the greater elements down one step. All the elements that are equal to the pivot are also highlighted with red colour. Three smaller and thinner `SmoothRectangles` surround the subsequences. Move the subsequences back up at the level of the sequence s. When it is time to sort a subsequence, move it down one step and give it a thick `SmoothRectangle` surrounding. Repeat the process until the algorithm has finished.

The movement speed of the sequences (`speed1`) and the movement speed of the elements (`speed2`) may be adjusted separately in the preference window. The pivot

choice strategy, and the several different pauses (in animation mode; the algorithm halts in step by step mode) may also be set here. The pauses occur:

- Before the pivot is picked (`pause1`).
- After the pivot is picked (`pause2`).
- Before the left sequence is sorted (`pause3`).
- Before the right sequence is sorted (`pause4`).
- Before combining the subsequences (`pause5`).
- Before moving the combined sequence back (`pause6`).

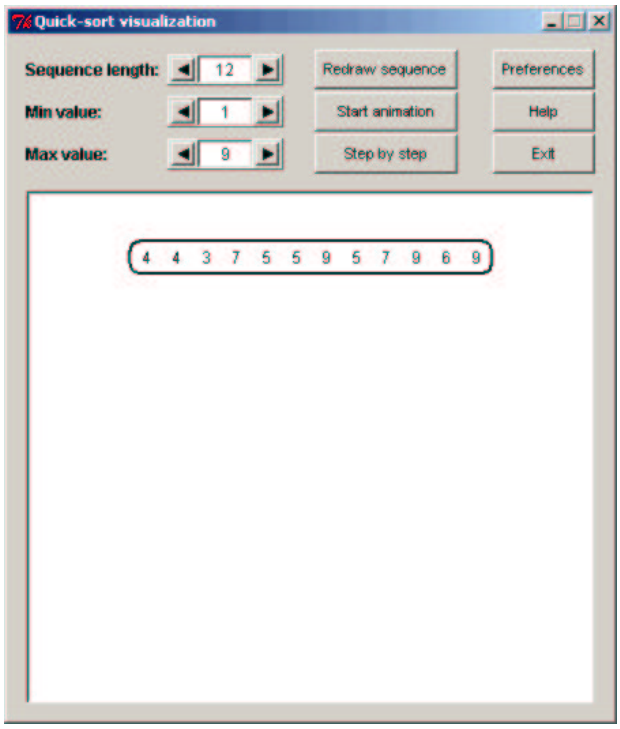


Figure 15.13: Before starting the algorithm.

15.7.4 Code

```
#!/bin/sh
""":"
exec python $0 ${1+"$@"}
"""

# Make sure the modules are found
import sys
sys.path[:0] = ['../']

# This will enable the program to be run from idle
import os
```

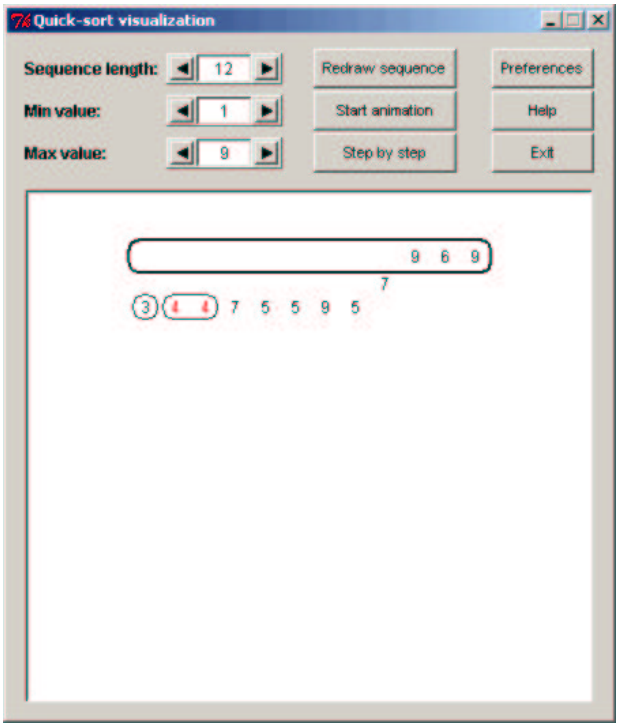


Figure 15.14: The original sequence is partitioned into three new sequences. The pivot is 4.

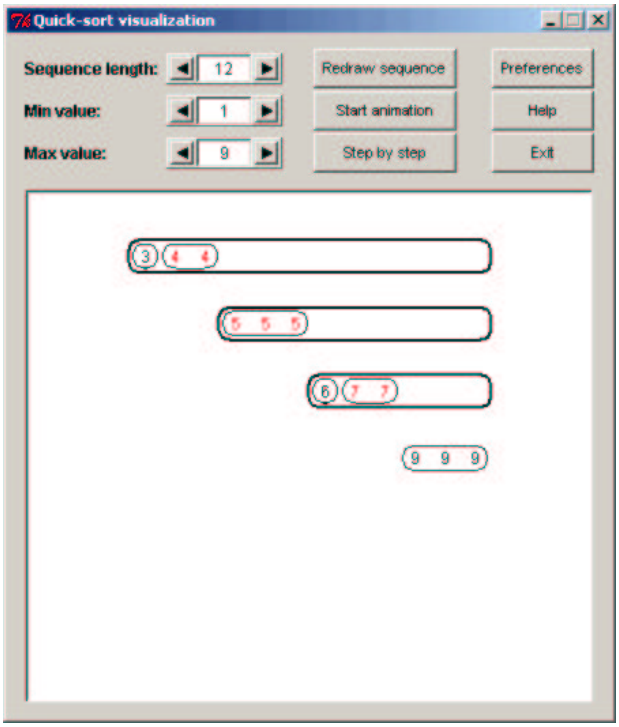


Figure 15.15: The individually sorted subsequences are joined to produce the sorted sequence.

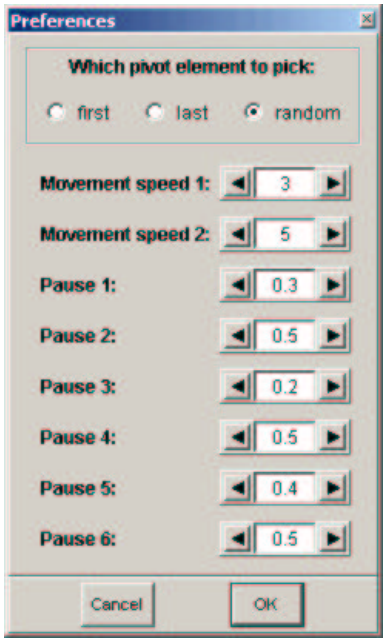


Figure 15.16: The preference window.

```
dirname,basename = os.path.split(sys.argv[0])
os.chdir(os.path.normpath(dirname))

import Tkinter
import Pmw
import whrandom
from sorting import baseclasses
from modules import dialogs


class Gui(baseclasses.BaseGui):
    def __init__(self):
        # Path to the reference page
        self.refPage = '../doc/' + 'applications_sorting_quick.html'

        self.x0 = 50                # Start position of sequence (x-coordinate)
        self.y0 = 50                # Start position of sequence (y-coordinate)
        self.dy = 50                # Distance between each sequence level
        self.seqValue = (10, 2, 14) # Sequence length (default, min, max)
        self.minValue = (1, -9, 8)  # Element min value (default, min, max)
        self.maxValue = (9, -8, 9)  # Element max value (default, min, max)
        self.canvasHeight = 380

        if os.name == 'nt':
            self.dx = 22            # Distance between elements in sequence
            self.canvasWidth = 420
            self.speedFactor = 6000
        else:
            self.dx = 32            # Distance between elements in sequence
            self.canvasWidth = 540
            self.speedFactor = 2000
```

```
        # Initialise class to handle algorithm
        self.sort = Sort()

        # Initialise base class
        baseclasses.BaseGui.__init__(self, root)
#----

class Sort:
    def __init__(self):
        self.sequence = None

    def redraw_sequence(self, x0, y0, dx, dy, length, min, max):
        self.y0 = y0
        self.dx = dx
        self.dy = dy
        elements = []
        for i in range(length):
            value = whrandom.randint(min, max)
            x = x0+i*dx
            y = y0
            element = baseclasses.Element(gui.canvas, value, x, y, dx, dy)
            elements.append(element)

        self.sequence = baseclasses.Sequence(elements)
        self.sequence.draw()
        self.sequence.highlight()

    def start_algorithm(self):
        self.quicksort(self.sequence, justStarted = 1)

    def quicksort(self, sequence, justStarted = 0):
        if len(sequence.elements) > 1:
            sequence.highlight()

            if not justStarted:
                gui.flow.wait_or_next_step(seconds = prefs.pause1)

            pivot = sequence.choose_pivot(prefs)
            pivot.highlight()
            gui.flow.wait_or_next_step(seconds = prefs.pause2)

            seq1,seq2,seq3 = self.divide_and_regroup(sequence.elements, pivot)
            gui.move_sequences((seq1,seq2,seq3), -.75, prefs.speed1)

            if len(seq1.elements):
                gui.flow.wait_or_next_step(seconds = prefs.pause3)
                gui.move_sequence(seq1, 1, prefs.speed1)
                self.quicksort(seq1)

            if len(seq3.elements):
                gui.flow.wait_or_next_step(seconds = prefs.pause4)
                gui.move_sequence(seq3, 1, prefs.speed1)
                self.quicksort(seq3)

            if len(sequence.elements)>1:
                gui.flow.wait_or_next_step(seconds = prefs.pause5)

            sequence.elements = seq1.elements + seq2.elements + seq3.elements
            gui.canvas.delete(seq1.tag, seq2.tag, seq3.tag)
```

```
            for element in seq2.elements:
                element.remove_highlight()

        if sequence.elements[0].y > self.y0:
            sequence.highlight(small = 1)
            if len(sequence.elements) > 1:
                gui.flow.wait_or_next_step(seconds = prefs.pause6)
            gui.move_sequence(sequence, -1, prefs.speed1)
        else:
            sequence.highlight()
            gui.display_end_message(135)

    def divide_and_regroup(self, elements, pivot):
        e1 = []
        e2 = []
        e3 = []
        x = elements[0].x
        y = elements[0].y + 3*self.dy/4

        for element in elements:
            if element.value < pivot.value:
                e1,x = self.append_and_move(element, e1, x, y)
        seq1 = baseclasses.Sequence(e1)
        seq1.highlight(small = 1)

        for element in elements:
            if element.value == pivot.value:
                e2,x = self.append_and_move(element, e2, x, y)
                element.highlight()
        seq2 = baseclasses.Sequence(e2)
        seq2.highlight(small = 1)

        for element in elements:
            if element.value > pivot.value:
                e3,x = self.append_and_move(element, e3, x, y)
        seq3 = baseclasses.Sequence(e3)
        seq3.highlight(small = 1)

        return seq1,seq2,seq3

    def append_and_move(self, element, elements, x, y):
        elements.append(element)
        gui.move_element(element, x, y, prefs.speed2)
        gui.canvas.resizescrollregion()
        element.x = x
        element.y = y
        x = x + self.dx
        return elements,x
#----

class Prefs(dialogs.BasePrefs):
    def __init__(self, gui):
        dialogs.BasePrefs.__init__(self, gui.balloon)

        self.pivot = 'random' # How to pick the pivot ('first','last','random')
        self.speed1 = 5   # Sequence movement speed (1:fast, 10:slow)
        self.speed2 = 3   # Element movement speed (during dividing)
        self.pause1 = 0.5 # Pause before picking the pivot (in seconds)
```

```
        self.pause2 = 0.8 # Pause after picking the pivot
        self.pause3 = 0.5 # Pause before sorting the left sequence
        self.pause4 = 0.5 # Pause before sorting the right sequence
        self.pause5 = 0.5 # Pause before combining the subsequences
        self.pause6 = 0.5 # Pause before moving combined sequence back

    def launch_window(self):
        if os.name == 'nt':
            width = 240
            height = 390
        else:
            width = 260
            height = 425

        self.create_dialog(root, width = width, height = height)

        args = ('Which pivot element to pick:', ('first', 'last', 'random'), 8)
        self.radioSelect = self.create_radioselect(*args)
        self.radioSelect.invoke(self.pivot)

        list = (('Movement speed 1: ', self.speed1, 'integer', 1, 10,
                 'Sequence movement speed (1:fast, 10:slow)'),
                ('Movement speed 2: ', self.speed2, 'integer', 1, 10,
                 'Element movement speed (during dividing) (1:fast, 10:slow)'),
                ('Pause 1: ', self.pause1, 'real', 0.0, 10.0,
                 'Pause (in seconds) before picking the pivot'),
                ('Pause 2: ', self.pause2, 'real', 0.0, 10.0,
                 'Pause (in seconds) after picking the pivot'),
                ('Pause 3: ', self.pause3, 'real', 0.0, 10.0,
                 'Pause (in seconds) before sorting the left sequence'),
                ('Pause 4: ', self.pause4, 'real', 0.0, 10.0,
                 'Pause (in seconds) before sorting the right sequence'),
                ('Pause 5: ', self.pause5, 'real', 0.0, 10.0,
                 'Pause (in seconds) before combining the subsequences'),
                ('Pause 6: ', self.pause6, 'real', 0.0, 10.0,
                 'Pause (in seconds) before moving combined sequence back'))
        self.counters = self.create_counters(list)

    def close_window(self, button):
        if button == 'OK':
            self.pivot = self.radioSelect.getcurselection()
            self.speed1 = int(self.counters[0].get())
            self.speed2 = int(self.counters[1].get())
            self.pause1 = float(self.counters[2].get())
            self.pause2 = float(self.counters[3].get())
            self.pause3 = float(self.counters[4].get())
            self.pause4 = float(self.counters[5].get())
            self.pause5 = float(self.counters[6].get())
            self.pause6 = float(self.counters[7].get())
        self.dialog.destroy()
        gui.prefsBtn.configure(state = 'normal')
#----

root = Tkinter.Tk()
root.title('Quick-sort visualization')

if os.name in ('nt', 'posix'):
    root.option_readfile('fontsAndColors.txt')
else:
```

```
    Pmw.initialise(root, fontScheme = 'pmw1')

gui = Gui()
prefs = Prefs(gui)
gui.startup(prefs)

root.resizable(0,0)
root.mainloop()
```

15.8 Shell-sort

15.8.1 About shell-sort

Shell-sort has a lot in common with insertion-sort (section 15.5). Both algorithms solve a sequence by swapping element positions. Unlike insertion-sort, which only compares adjacent elements, shell-sort starts by comparing distant elements. The running time of the algorithm will in general depend strongly on the choice of this distance.

More information on shell-sort may be found in [11].

15.8.2 The algorithm

Let's assume that we have a sequence of positive integers, $h_1, h_2, \ldots, h_k$, called the increment sequence. (This is in addition to the sequence s to be sorted.) The increment sequence may be arbitrary as long as the last element, h_k, equals 1. We start the algorithm by comparing elements in s that are spaced h_1 apart. Swap the positions of the elements when necessary. We now say that the sequence s is h_1-sorted. Continue doing this for each element in the increment sequence. Before starting the h_k-sort, which is equivalent to the insertion-sort algorithm, the sequence s should be almost sorted, making the last pass run quickly.

15.8.3 The illustration

The shell-sort illustration is almost identical to the insertion-sort (section 15.5) illustration. The only difference is the increment sequence shown in the upper right corner of the canvas. The active increment element will be highlighted in red.

In the preference window the "Increment sequence:" entry field is the only addition.

15.8.4 Code

```
#!/bin/sh
""":"
exec python $0 ${1+"$@"}
"""

# Make sure the modules are found
import sys
sys.path[:0] = ['../']

# This will enable the program to be run from idle
import os
dirname,basename = os.path.split(sys.argv[0])
os.chdir(os.path.normpath(dirname))
```

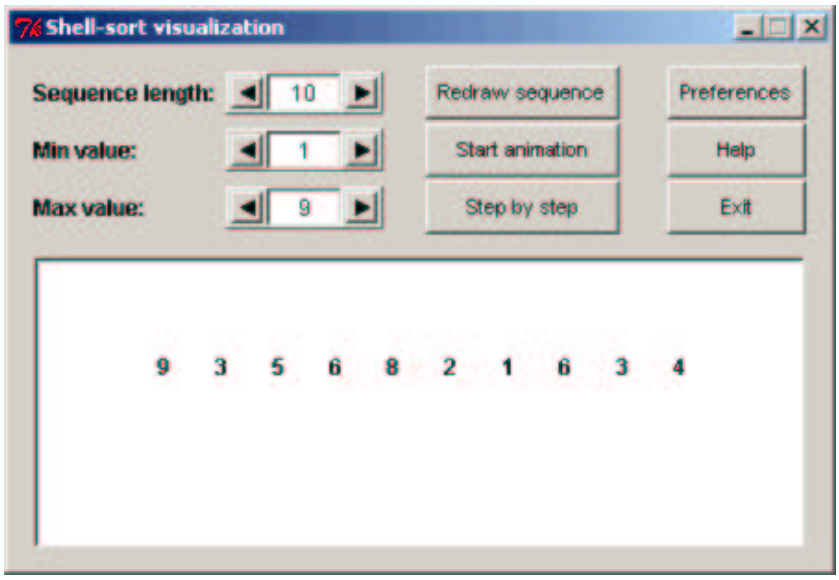


Figure 15.17: Before starting the algorithm.

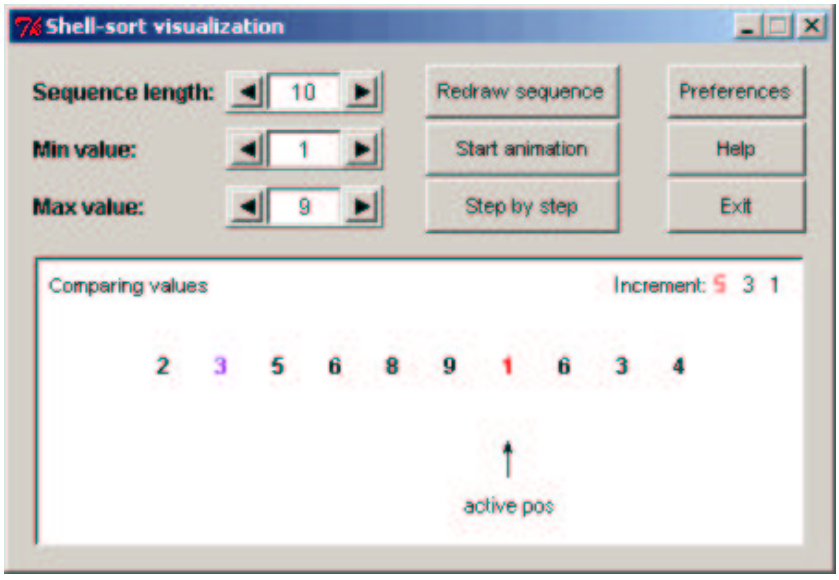


Figure 15.18: Comparing two elements spaced 5 elements apart.

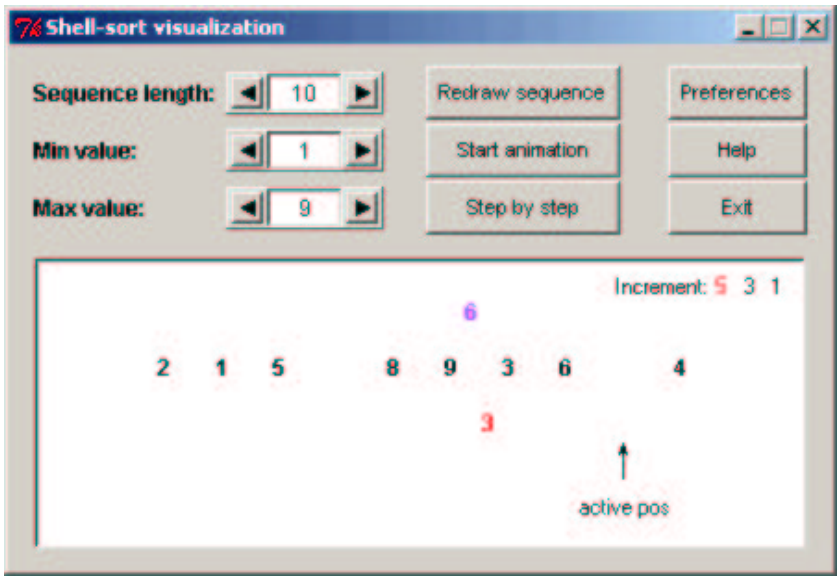


Figure 15.19: Swapping two elements' positions.

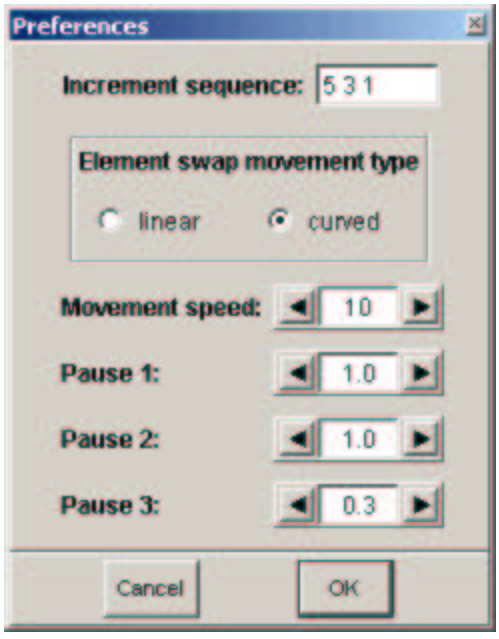


Figure 15.20: The preference window.

```
import Tkinter
import Pmw
import whrandom
from sorting import baseclasses
from modules import dialogs


class Gui(baseclasses.BaseGui):
    def __init__(self):
        # Path to the reference page
        self.refPage = '../doc/' + 'applications_sorting_shell.html'

        self.x0 = 50                # Start position of sequence (x-coordinate)
        self.y0 = 60                # Start position of sequence (y-coordinate)
        self.dy = 40                # Height when swapping elements
        self.seqValue = (10, 3, 12) # Sequence length (default, min, max)
        self.minValue = (1, -9, 8)  # Element min value (default, min, max)
        self.maxValue = (9, -8, 9)  # Element max value (default, min, max)
        self.canvasHeight = 160

        if os.name == 'nt':
            self.dx = 31            # Distance between elements in sequence
            self.canvasWidth = 420
            self.speedFactor = 6000
        else:
            self.dx = 40            # Distance between elements in sequence
            self.canvasWidth = 540
            self.speedFactor = 2000

        # Initialise class to handle algorithm
        self.sort = Sort()

        # Initialise base class
        baseclasses.BaseGui.__init__(self, root)
#----

class Sort:
    def __init__(self):
        self.elements = []

    def redraw_sequence(self, x0, y0, dx, dy, length, min, max):
        self.elements = []
        for i in range(length):
            value = whrandom.randint(min, max)
            element = baseclasses.Element(gui.canvas, value, x0+i*dx, y0)
            element.draw(font = ('Helvetica', 9, 'bold'))
            self.elements.append(element)

    def start_algorithm(self):
        self.shellsort()

    def shellsort(self):
        gui.display_incr_seq(prefs.incrSeq)

        for increment in prefs.incrSeq:
            item = 'incr_' + str(prefs.incrSeq.index(increment))
            gui.canvas.itemconfigure(item, fill = 'Red')

            for element in self.elements[increment:]:
```

```
                gui.draw_active_pos_arrow(element)
                element.highlight(font = ('Helvetica', 9, 'bold'))
                gui.flow.wait_or_next_step(seconds = prefs.pause1)

                index = self.elements.index(element)
                while index - increment >= 0:
                    element2 = self.elements[index-increment]
                    kw = {'colour': 'Purple', 'font': ('Helvetica', 9, 'bold')}
                    element2.highlight(**kw)

                    message = 'Comparing values'
                    gui.display_message(message, pause = prefs.pause2)

                    gui.canvas.delete('message')

                    if element.value < element2.value:
                        args = (element, element2, prefs.swapType, prefs.speed)
                        element.x,element2.x = gui.swap_elements(*args)

                        self.elements[index] = element2
                        self.elements[index-increment] = element
                        index = index - increment
                    else:
                        index = 0

                    element2.remove_highlight()

                element.remove_highlight()
                gui.canvas.delete('arrow')
                gui.flow.wait_or_next_step(seconds = prefs.pause3)

            item = 'incr_' + str(prefs.incrSeq.index(increment))
            gui.canvas.itemconfigure(item, fill = 'Black')

        gui.display_end_message(35, font = ('Helvetica', 10, 'bold'))
#----


class Prefs(dialogs.BasePrefs):
    def __init__(self, gui):
        dialogs.BasePrefs.__init__(self, gui.balloon)

        self.incrSeq = [5, 3, 1]  # The increment sequence
        self.swapType = 'curved'  # Swap movement type ('linear','curved')
        self.speed = 3            # Element movement speed
        self.pause1 = 1.0         # Pause after highlighting active element
        self.pause2 = 1.0         # Pause when comparing elements
        self.pause3 = 0.3         # Pause when an element has finished swapping

    def launch_window(self):
        if os.name == 'nt':
            width = 240
            height = 292
        else:
            width = 250
            height = 318
        self.create_dialog(root, width = width, height = height)

        value = ''
```

```
        for n in self.incrSeq:
            value = value + str(n) + ' '
        args = ('Increment sequence: ', 8, value, 10)
        self.entryField = self.create_entryfield(*args)

        args = ('Element swap movement type', ('linear', 'curved'), 5)
        self.radioSelect = self.create_radioselect(*args)
        self.radioSelect.invoke(self.swapType)

        list = (('Movement speed: ', self.speed, 'integer', 1, 10,
                 'Element movement speed (1:fast, 10:slow)'),
                ('Pause 1: ', self.pause1, 'real', 0.0, 10.0,
                 'Pause (in seconds) after highlighting active element'),
                ('Pause 2: ', self.pause2, 'real', 0.0, 10.0,
                 'Pause (in seconds) when comparing elements'),
                ('Pause 3: ', self.pause3, 'real', 0.0, 10.0,
                 'Pause (in seconds) when an element has finished swapping'))
        self.counters = self.create_counters(list)

    def close_window(self, button):
        if button == 'OK':
            self.incrSeq = map(int, self.entryField.get().split())
            self.swapType = self.radioSelect.getcurselection()
            self.speed = int(self.counters[0].get())
            self.pause1 = float(self.counters[1].get())
            self.pause2 = float(self.counters[2].get())
            self.pause3 = float(self.counters[3].get())
        self.dialog.destroy()
        gui.prefsBtn.configure(state = 'normal')
#----

root = Tkinter.Tk()
root.title('Shell-sort visualization')

if  os.name in ('nt', 'posix'):
    root.option_readfile('fontsAndColors.txt')
else:
    Pmw.initialise(root, fontScheme = 'pmw1')

gui = Gui()
prefs = Prefs(gui)
gui.startup(prefs)

root.resizable(0,0)
root.mainloop()
```

Chapter 16

Vibrations in mechanical systems

16.1 Introduction

We are going to look at two applications involving vibrations in mechanical systems. Both systems are build around a box connected to a spring. The first case (free vibrations (section 16.4) involves only forces that are internal to the system, while an external force is applied in the second case (forced vibrations (section 16.5)).

Both applications have to solve a second order linear differential equation. The calculations are done in the `System` class (section 16.2).

Some of the GUI elements and some of the handling of visual properties are common for both applications. A base class (section 16.3) has therefore been created to limit code repetition.

The actual drawing of the spring is managed by the `Spring` class from the `widgets` module (chapter 10).

16.2 System class

16.2.1 Introduction

This is where the mathematical model of the spring mass system is defined and managed. The base class `Vibrations` and its derived classes `FreeVibrations` and `Forced-Vibrations`, handle all parameter values and calculates the solution to the system's second order linear differential equation.

The deduction of the equations from the physical situations is done in the application sections (16.4 and 16.5).

16.2.2 The base class: Vibrations

Functions:

- `__init__()` - Define all the system's parameters. Use the function `new_parameter-_value`, found in the derived classes, to change them. Increase or decrease the value of the attribute `dt` to speed up or slow down the animation.

- `advance_one_step()` - Increase the time step t. Calculate the value $y = y(t)$. Return both numbers.

- `solution()` - Find the approriate solution as a text string. Use the class `Function` from the `functions` module (chapter 7) to create a solution object `y`. Return a sample of values $\big(t, y(t)\big)$ from the interval $(0, t_{\max})$.
- `state_of_system()` - Find the state of the system. The state of the system and the solution equation depends on the value of $q^2 - 4k$. Return a text string describing the state.
- `_normal_damped()` - Create and return the solution string for the normal damped system state ($q^2 - 4k < 0$). Find the particular solution string by calling the derived class' `particular` function.
- `_critically_damped()` - Create and return the solution string for the critically damped system state ($q^2 - 4k = 0$). Find the particular solution string by calling the derived class' `particular` function.
- `_overdamped()` - Create and return the solution string for the overdamped system state ($q^2 - 4k > 0$). Find the particular solution string by calling the derived class' `particular` function.

16.2.3 The derived classes FreeVibrations and ForcedVibrations

Functions:

- `new_parameter_values(*args)` - Change the parameter values of the system. The `FreeVibrations` class does not use the values $y0$, A and ω. A call to the `solution` function should be made to incorporate the new values into the solution object `y`.
- `_particular()` - Create and return the particular solution string for the application. In the `FreeVibrations` case the particular solution equals zero, i.e. the differential equation is homogeneous.

16.2.4 Code

```
# Defines and calculates the mathematical model of a vibrating spring
# class FreeVibrations: without external force
# class ForcedVibrations: with external force

# Make sure the modules are found
import sys
sys.path[:0] = ['../']

from modules import functions
from math import *

class Vibrations:
    def __init__(self):
        self.q = 1.0        # The damping force
        self.k = 6.0        # The stiffness of the spring

        self.y = None       # Solution (displacement of spring)
        self.y0 = 1.0       # Initial displacement
        self.v0 = 0.0       # Initial velocity (in y-direction)
```

```
        self.t = 0          # Current simulation time step
        self.t_max = 10.0   # When to end the simulation
        self.dt = .1        # How much to advance the simulation in each step
        self.samples = 8*self.t_max # How refined the movement should be

        self.A = 0.5        # Amplitude of external force
        self.omega = 1.0    # Period of external force
        self.u = 1.0        # Velocity of object (in x-direction)


    # Advance the movement one step and return (t,y) value
    def advance_one_step(self):
        self.t = self.t + self.dt
        y = self.y(self.t)
        return self.t,y


    # Find system state, call appropriate solution function,
    # and return calculated solution
    def solution(self):
        state = self.state_of_system()
        if state == 'Normal damped motion':
            solution = self._normal_damped()
        if state == 'Critically damped':
            solution = self._critically_damped()
        if state == 'Overdamped':
            solution = self._overdamped()

        self.y = functions.Function(solution, 0.0, self.t_max,
                variable = 't',
                samples = self.samples)
        return self.y.calculate_values()


    # Return the state of the system
    def state_of_system(self):
        parameter = self.q*self.q - 4*self.k
        if parameter < 0:
            return 'Normal damped motion'
        if parameter == 0:
            return 'Critically damped'
        if parameter > 0:
            return 'Overdamped'


    # Return solution to normal damped (underdamped) case
    def _normal_damped(self):
        F,G,yp = self._particular()
        a = -self.q/2.0
        b = sqrt(4*self.k - self.q*self.q)/2.0
        C1 = self.y0 - G
        C2 = float(self.v0 - a*C1 - self.omega*F)/b

        # Homogeneous solution: yh = e^(at)*(C1*cos(bt) + C2*sin(bt))
        yh = 'e**(' + str(a) + '*t)*(' + str(C1) + '*cos(' + str(b) +\
             '*t)+' + str(C2) + '*sin(' + str(b) + '*t))'
        return yh + '+' + yp
```

```
    # Return solution to critically damped case
    def _critically_damped(self):
        F,G,yp = self._particular()
        r = -self.q/2.0
        C1 = self.y0 - G
        C2 = self.v0 - C1*r - self.omega*F

        # Homogeneous solution: yh = C1*e^(rt) + C2*t*e^(rt)
        yh = str(C1) + '*e**(' + str(r) + '*t)+' + str(C2) + '*t*e**(' +\
             str(r) + '*t)'
        return yh + '+' + yp


    # Return solution to the overdamped case
    def _overdamped(self):
        F,G,yp = self._particular()
        r1 = -(self.q/2.0) + sqrt(self.q*self.q - 4*self.k)/2.0
        r2 = -(self.q/2.0) - sqrt(self.q*self.q - 4*self.k)/2.0
        C1 = (self.v0 - self.y0*(r2 - G) - self.omega*F)/float(r1 - r2)
        C2 = self.y0 - C1 - G

        # Homogeneous solution: yh = C1*e^(r1t) + C2*e^(r2t)
        yh = str(C1) + '*e**(' + str(r1) + '*t)+' + str(C2) +\
             '*e**(' + str(r2) + '*t)'
        return yh + '+' + yp
#----


class FreeVibrations(Vibrations):
    def __init__(self):
        Vibrations.__init__(self)

    # Set new parameter values
    def new_parameter_values(self, q, k, t_max):
        self.q = q
        self.k = k
        self.t_max = t_max
        self.samples = 8*t_max
        if self.samples == 0:
            self.samples = 2

    # Return particular solution
    def _particular(self):
        return 0,0,'0'
#----


class ForcedVibrations(Vibrations):
    def __init__(self):
        Vibrations.__init__(self)

    # Set new parameter values
    def new_parameter_values(self, y0, q, k, t_max, A, omega):
        self.y0 = y0
        self.q = q
        self.k = k
        self.t_max = t_max
        self.A = A
        self.omega = omega
```

```
        self.samples = 8*t_max
        if self.samples == 0:
            self.samples = 2

    # Return particular solution
    def _particular(self):
        a = self.k - self.omega*self.omega
        b = self.q*self.omega
        F = self.A*a/float(a*a + b*b)
        G = -self.A*b/float(a*a + b*b)

        # Particular solution: yp = F*sin(omega*t) + G*cos(omega*t)
        yp = str(F) + '*sin(' + str(self.omega) + '*t) +' +\
             str(G) + '*cos(' + str(self.omega) + '*t)'
        return F,G,yp
```

16.3 Base gui class

This is a base class for the free vibrations application (section 16.4) and the forced vibrations application (section 16.5). A subclass should be created with the following mandatory attributes and functions:

- `system` - An object of type `FreeVibrations` or `ForcedVibrations` (see the system class (section 16.2)).
- `profileCanvas` - A *Tkinter.Canvas* object.
- `new_parameter_value()` - Handle scale changes.
- `draw_spring(t, y)` - Draw the spring (and possible attached objects) at time level `t` with displacement `y`.

The functions:

- `create_canvas(width, height)` - Convenience function used to create a *Tkinter.Canvas*.
- `create_scale(label, from_, to, digits, ticint, res, init)` - Convenience function used to create a horizontal *Tkinter.Scale*. The function `new_parameter_value` is assumed available.
- `animate_spring()` - This will run an animation of the solution of the system. The function `draw_spring` and the attributes `system` and `profileCanvas` are assumed available.
- `draw_profile()` - Draw a profile of the spring displacement solution in the profile canvas. The attributes `system` and `profileCanvas` are assumed available.
- `draw_profileCanvas_stuff()` - Draw a text label and two axes (t and y) in the profile canvas. The attribute `profileCanvas` is assumed available.
- `find_canvas_values(values, base_y = 70)` - Used to convert coordinate values to canvas values. The attribute `system` is assumed available.

- `help()` - Display reference page.
- `exit()` - Exit application.
- `_draw_marker(t, y)` - This is a private function used by the `animate_spring` function. It will draw the marker (a filled circle) on top of the spring displacement profile at (profile canvas) coordinates (t, y).

```
import Tkinter
import os
import thread

class BaseGui:
    def __init__(self, master):
        self.master = master
        master.bind('<q>', self.exit)
        master.bind('<h>', self.help)


    # Create and return a Tkinter.Canvas
    def create_canvas(self, width, height):
        canvas = Tkinter.Canvas(self.master,
                background = 'White',
                borderwidth = 2,
                height = height,
                relief = 'sunken',
                width = width)
        return canvas


    # Create, initialise and return a Tkinter.Scale
    def create_scale(self, label, from_, to, digits, tickint, res, init):
        scale = Tkinter.Scale(self.master,
                command = self.new_parameter_value,
                digits = digits,
                from_ = from_,
                label = label,
                length = 225,
                orient = 'horizontal',
                resolution = res,
                tickinterval = tickint,
                to = to,
                troughcolor = 'White')
        scale.set(init)
        return scale

    # Animate the spring
    def animate_spring(self):
        t = 0
        t_max = self.system.t_max
        while t <= t_max:
            t,y = self.system.advance_one_step()
            self.draw_spring(t, y)
            self._draw_marker(t, y)
            self.master.update()
        self.system.t = 0
        self.profileCanvas.delete('marker')
```

```
# Calculate and draw the spring displacement profile
def draw_profile(self):
    # Get the system's solution
    values = self.system.solution()

    # Calculate the corresponding canvas values
    cvalues = self.find_canvas_values(values)

    # Draw the profile
    self.profileCanvas.delete('profile')
    self.profileCanvas.create_line(cvalues, fill = 'Blue', tags = 'profile')


# Draw the coordinate axis and the label in the profileCanvas
def draw_profileCanvas_stuff(self):
    kw = {'anchor': 'e', 'text': 'Spring displacement profile',
          'font': 'Helvetica 9 bold'}
    self.profileCanvas.create_text(500,10, **kw)
    kw = {'arrowshape': (6,8,2)}
    self.profileCanvas.create_line(25,5, 25,130, arrow = 'first', **kw)
    self.profileCanvas.create_line(10,70, 490,70, arrow = 'last', **kw)
    font = 'Helvetica 10'
    self.profileCanvas.create_text(15, 13, text = 'y', font = font)
    self.profileCanvas.create_text(495, 70, text = 't', font = font)


# Calculate suitable canvas values
def find_canvas_values(self, values, base_y = 70):
    cvalues = []
    for t,y in values:
        try:
            cx = 25 + t*450.0/self.system.t_max
        except:
            cx = 25
        cy = base_y - y*45
        cvalues.extend((cx,cy))
    return cvalues


# Display the reference page
def help(self, event = None):
    if os.name == 'nt':
        app = os.path.normpath(self.refPage)
    else:
        app = 'netscape ' + os.path.normpath(self.refPage) + '&'
    thread.start_new_thread(os.system, (app,))


# Close window and exit program
def exit(self, event = None):
    self.master.destroy()


# Draw the marker (a filled circle) along the spring displacement profile
def _draw_marker(self, t, y):
    # Calculate the corresponding canvas values
    cx,cy = self.find_canvas_values(((t,y),))

    # Draw the marker
```

```
self.profileCanvas.delete('marker')
kw = {'fill': 'Red', 'tags': 'marker'}
self.profileCanvas.create_oval(cx-5,cy-5, cx+5,cy+5, **kw)
```

16.4 Vibrating spring without external force (free vibrations)

16.4.1 The physical problem

Consider a box of mass m attached to a spring. We assume that the spring is without mass and that it is attached to the ceiling (see figure 16.1). The spring exerts no force on the box when the box is in equilibrium. We are going to look at what happens when the box is displaced a distance y_0 away from this position. The initial velocity of the box is given by v_0.

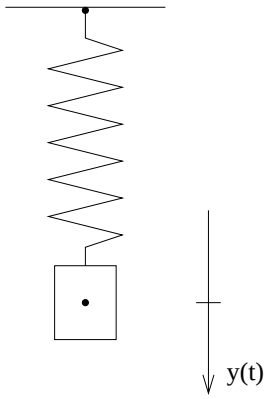


Figure 16.1: Sketch of a spring-mass system.

16.4.2 The mathematical model

The force of the spring is, according to Hooke's law, proportional to the displacement from equilibrium:

$$F_k = -ky,$$

where k is a measure of the stiffness of the spring, and $y = y(t)$ is the spring's position at time t. The system is in equilibrium when $y = 0$. We further assume the existence of a damping force proportional to the velocity:

$$F_q = -qy',$$

where q is a measure of the resistance in the medium (the air, for instance) and $y' = y'(t)$ is the velocity of the spring/box. By Newton's second law of motion

$$F = F_k + F_q = ma,$$

we have

$$my'' = -ky - qy', \quad k > 0, \quad q > 0.$$

To keep things simple, we assume that the mass of the box equals 1, so that the equation governing the motion of the spring/box can be written

$$y'' + qy' + ky = 0,$$

with initial conditions

$$y(0) = y_0, \quad y'(0) = v_0.$$

This is a second order homogeneous differential equation with constant coefficients which may be solved by looking at the roots of the auxiliary equation

$$r^2 + qr + k = 0,$$

$$r = -\frac{q}{2} \pm \frac{\sqrt{q^2 - 4k}}{2}.$$

The value $q^2 - 4k$ determines the type of the roots. Different types of roots leads to different solutions to the differential equation. The deduction of the solutions may be found in a textbook on differential equations, for example [9]. We will only state the results here.

Case 1: underdamped (normal) motion: $\mathbf{q^2 - 4k < 0}$
The auxiliary equation has two distinct complex roots

$$r = a \pm ib,$$

and the solution is given by

$$y(t) = e^{at}\big(C_1 \cos bt + C_2 \sin bt\big),$$

where the constants C_1 and C_2 are determined by the initial conditions

$$C_1 = y_0, \quad C_2 = \frac{v_0 - aC_1}{b}.$$

Case 2: critically damped motion: $\mathbf{q^2 - 4k = 0}$
The auxiliary equation has one real root

$$r = -q/2,$$

and the solution is given by

$$y(t) = C_1 e^{rt} + C_2 t e^{rt},$$

where the constants C_1 and C_2 are determined by the initial conditions

$$C_1 = y_0, \quad C_2 = v_0 - C_1 r.$$

Case 3: overdamped motion: $\mathbf{q^2 - 4k > 0}$
The auxiliary equation has two distinct real roots r_1 and r_2 and the solution is given by

$$y(t) = C_1 e^{r_1 t} + C_2 e^{r_2 t},$$

where the constants C_1 and C_2 are determined by the initial conditions

$$C_1 = \frac{v_0 - y_0 r_2}{r_1 - r_2}, \quad C_2 = y_0 - C_1.$$

16.4.3 The illustration

The basic idea is very simple. We need a canvas two draw the spring/box system, and another canvas to draw the spring movement profile. (The spring movement canvas could instead be created as a *BLT.Graph*. See the installation section (2.3) for a reason to avoid using BLT when other options are possible.) When the spring is released the spring/box begins to move. A red marker (a filled circle) follows the movement along the profile graph. The animation ends after an adjustable time period has expired.

16.4.4 The user interface

The initial displacement of the spring/box may be determined by pressing the left mouse button in the spring/box canvas and moving the mouse up or down while pressing the button. The positions $y = -1$, $y = 0$ and $y = 1$, i.e. the minimal displacement, equilibrium, and maximum displacement, is achievd by pressing the left mouse button on the corresponding text objects. Deselect the checkbutton "Draw constant y-lines" to remove the text objects and their constant y-lines.

Apart from the initial displacement, there are three adjustable parameters in the system: q (the damping force), k (the stiffness of the spring), and the length of the animation. (The initial velocity, v_0, has not been made available in this application.) A *Tkinter.Scale* is used to change their values. If either one of the scales or the initial displacement is adjusted, the spring movement graph is changed accordingly.

The state of the system and the value of $q^2 - 4k$, are displayed at the top of the spring/box canvas.

The spring movement animation is started by pressing the "Release spring" button. The "Help" button displays the reference page (a *html* version of this section), and the "Exit" button exits the application.

16.4.5 Code

```
#!/bin/sh
""":"
exec python $0 ${1+"$@"}
"""

# Make sure the modules are found
import sys
sys.path[:0] = ['../']

# This will enable the program to be run from idle
import os
dirname,basename = os.path.split(sys.argv[0])
os.chdir(os.path.normpath(dirname))

import Tkinter
import Pmw
from modules import misc
from modules import widgets
from vibrations import system
from vibrations import basegui
from math import *


class Gui(basegui.BaseGui):
```

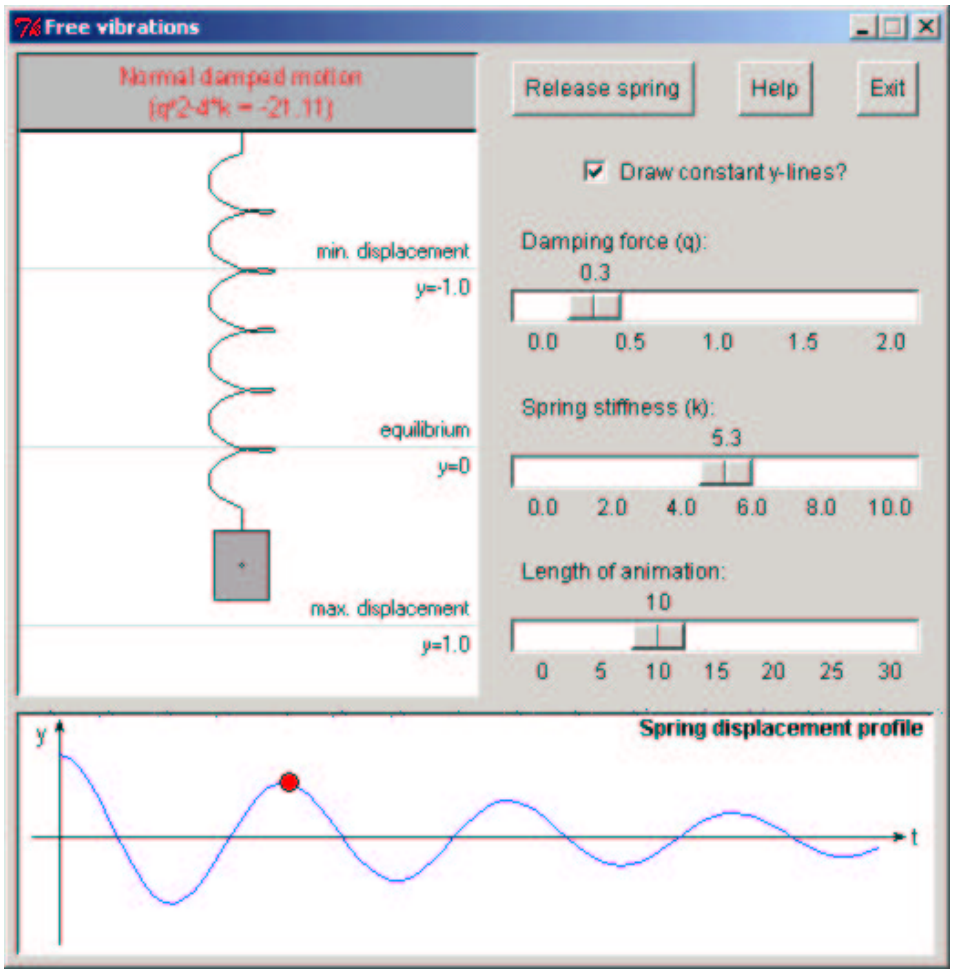


Figure 16.2: Snapshot of the application in action.

```
    """Handle interface and user interaction"""

    def __init__(self):
        # Path to the reference page
        self.refPage = '../doc/' + 'applications_vibrations_free.html'

        # Initialise base class
        basegui.BaseGui.__init__(self, root)

        # Create the main canvas
        self.canvas = self.create_canvas(250, 350)
        self.canvas.grid(row = 0, rowspan = 5, column = 0, padx = 2, pady = 2)
        self.canvas.bind('<Button-1>', self.LMB_pressed)
        self.canvas.bind('<Button1-Motion>', self.LMB_motion)

        # Draw the roof in the main canvas
        self.canvas.create_rectangle(-5,-5, 405,45, width = 2, fill = 'gray75')

        # Create the buttons
        if os.name == 'nt':
            padx = 12
        else:
            padx = 0
        self.buttonBox = Pmw.ButtonBox(root, padx = padx)
        self.buttonBox.add('Release spring', command = self.animate_spring)
        self.buttonBox.add('Help', command = self.help)
        self.buttonBox.add('Exit', command = self.exit)
        self.buttonBox.grid(row = 0, column = 1, padx = 2, pady = 2)

        # Create the checkbutton
        self.linesVar = Tkinter.IntVar()
        self.linesVar.set(1)
        self.linesCB = Tkinter.Checkbutton(root,
                command = self.draw_constant_y_lines,
                text = 'Draw constant y-lines?',
                variable = self.linesVar)
        self.linesCB.grid(row = 1, column = 1, padx = 2, pady = 2)

        # Create the scales
        list = (('Damping force (q):', 0.0, 2.0, 2, .5, .1, 0.5),
                ('Spring stiffness (k):', 0.0, 10.0, 3, 2, .1, 5.0),
                ('Length of animation:', 0, 30, 2, 5, 1, 6))
        self.scales = []
        for args in list:
            scale = self.create_scale(*args)
            scale.grid(row = len(self.scales)+2, column = 1, padx = 2)
            self.scales.append(scale)

        # Create the spring displacement profile canvas
        self.profileCanvas = self.create_canvas(500, 130)
        self.profileCanvas.grid(row = 5, columnspan = 2, padx = 2, pady = 2)
        self.draw_profileCanvas_stuff()

        # Class FreeVibrations handles the mathematical model of the system
        self.system = system.FreeVibrations()

        # Create the spring
        self.spring = widgets.Spring(self.canvas, 125, 45, 154)
```

```
        self.draw_constant_y_lines()


    # Draw the spring at the given displacement
    def draw_spring(self, t, displacement, hooks = 0):
        self.canvas.delete('spring')

        # Draw spring
        self.spring.draw(displacement)

        # Draw the box at the end of the spring
        x,y = self.spring.get_end_position(displacement)
        kw = {'fill': 'DarkGray', 'tags': 'spring'}
        self.canvas.create_rectangle(x-15,y, x+15,y+38, **kw)
        self.canvas.create_oval(x-1,y+18, x+1,y+20, tags = 'spring')

        # Draw the hooks if necessary
        if hooks:
            kw = {'tags': ('spring', 'hook')}
            self.canvas.create_line(x-15,y+19, 0,y+19, **kw)
            self.canvas.create_line(x+15,y+19, 255,y+19, **kw)


    # Update all parameter values. Draw system state, spring and profile
    def new_parameter_value(self, dummy = 0):
        q = self.scales[0].get()
        k = self.scales[1].get()
        t_max = self.scales[2].get()

        self.system.new_parameter_values(q, k, t_max)

        # Draw system state
        self.canvas.delete('state')
        parameter = q*q - 4*k
        state = self.system.state_of_system()
        kw = {'text': state, 'fill': 'Red', 'font': 'Helvetica 10',
              'tags': 'state'}
        self.canvas.create_text(125, 15, **kw)
        kw['text'] = '(q^2-4*k = ' + str(parameter) + ')'
        self.canvas.create_text(125, 32, **kw)

        self.draw_spring(0, self.system.y0, hooks = 1)
        self.draw_profile()


    # The user is about to move the spring
    def LMB_pressed(self, event):
        self.y = event.y

    # Move the spring to a new displacement
    def LMB_motion(self, event):
        dy = event.y - self.y
        if dy > 100:
            dy = 100
        if dy < -100:
            dy = -100

        self.system.y0 = dy/100.0
        self.draw_spring(0, dy/100.0, hooks = 1)
```

```
        self.draw_profile()


    # Set the spring displacement to min, equilibrium or max
    def set_spring_position(self, event, text):
        self.system.y0 = float(text[2:])
        self.draw_spring(0, float(text[2:]), hooks = 1)
        self.draw_profile()


    # Draw the constant y-lines in the main canvas
    def draw_constant_y_lines(self):
        if not self.linesVar.get():
            self.canvas.delete('lines')
            return

        list = ((120, 'min. displacement','y=-1.0'),
                (218, 'equilibrium','y=0'),
                (316, 'max. displacement','y=1.0'))
        for y,text1,text2 in list:
            kw = {'fill': '#cccccc', 'tags': ('lines', text1)}
            self.canvas.create_line(0,y, 255,y, **kw)

            kw = {'anchor': 'e', 'text': text1, 'tags': ('lines', text1)}
            self.canvas.create_text(250,y-10, **kw)

            kw['text'] = text2
            self.canvas.create_text(250,y+10, **kw)

            cmd = misc.Command(self.set_spring_position, text2)
            self.canvas.tag_bind(text1, '<Button-1>', cmd)
#----

root = Tkinter.Tk()
root.title('Free vibrations')

if os.name in ('nt', 'posix'):
    root.option_readfile('fontsAndColors.txt')
else:
    Pmw.initialise(root, fontScheme = 'pmw1')

Gui()

root.resizable(0,0)
root.mainloop()
```

16.5 Vibrating spring with external force (forced vibrations)

16.5.1 The physical problem

Consider a car (of mass $m = 1$) driving down a corrugated road. To make things simple we are going to look at a car shaped like a box with only one wheel. The box and the wheel are connected by means of a spring (see figure 16.3). We further assume that the horizontal velocity (u) of the car is constant and that the road is continuous and periodic. The initial displacement of the spring from its equilibrium position is given

by y_0 and the initial vertical velocity v_0.

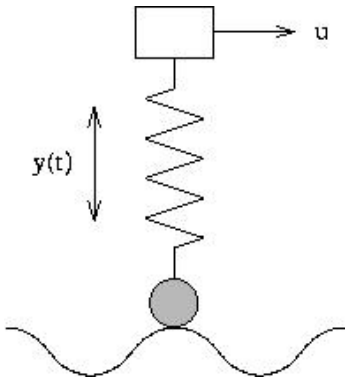


Figure 16.3: Sketch of the physical situation.

16.5.2 The mathematical model

Assume that the road can be expressed as

$$y = f(ut) = A\sin\omega t, \quad u = 1,$$

where we have set the constant horizontal velocity to 1. The road acts as an external force on the car and the differential equation deducted in the free vibrations application (section 16.4) now extends to

$$y'' + qy' + ky = A\sin\omega t.$$

The same initial conditions apply

$$y(0) = y_0, \quad y'(0) = v_0.$$

This equation is solved by adding a particular solution to the general solution of the homogeneous equation. Let us use

$$y_p(t) = F\sin\omega t + G\cos\omega t$$

as a trial solution. This leads to the following pair of equations

$$\begin{aligned} -F\omega^2 - qG\omega + kF &= A, \\ -G\omega^2 + qF\omega + kG &= 0. \end{aligned}$$

If we let

$$a = k - \omega^2, \quad b = q\omega,$$

the solution can be written as

$$\begin{aligned} F &= \frac{Aa}{a^2+b^2}, \\ G &= -\frac{Ab}{a^2+b^2}. \end{aligned}$$

The constants C_1 and C_2 from the solution of the homogeneous equation ($y_h(t)$) can now be calculated. They are

- The underdamped case:

$$C_1 = y_0 - G, \quad C_2 = \frac{v_0 - C_1 a - F\omega}{b}.$$

- The critically damped case:

$$C_1 = y_0 - G, \quad C_2 = v_0 - C_1 r - F\omega.$$

- The overdamped case:

$$C_1 = \frac{v_0 - y_0(r_2 - G) - F\omega}{r_1 - r_2}, \quad C_2 = y_0 - C_1 - G.$$

The general solution to the nonhomogeneous equation now becomes

$$y(t) = y_h(t) + y_p(t).$$

16.5.3 The illustration

This is very similar to the free vibrations (section 16.4) illustration. Two canvases of equal width are used, one on top of the other. The road and the car are drawn in the top canvas, while the spring movement profile is drawn in the bottom one. (The spring movement canvas could instead be created as a *BLT.Graph*. See the installation section (2.3) for a reason to avoid using BLT when other options are available.) The wheel of the car follows the same curve as the road from left to right. The red marker in the profile canvas moves from left to right at the same speed as the car. The animation stops when the car reaches the right edge of the canvas.

16.5.4 The user interface

There are six adjustable parameters in the system:

- The initial displacement: y_0
- The damping force: q
- The spring stiffness: k
- The length of the animation
- The amplitude of the road: A
- The period of the road: ω

(In fact, there are some more: the initial vertical velocity, the horizontal velocity and the mass of the car. They are not available in this application, but they could make a nice extension.)

The value of the above parameters can each be changed by means of a *Tkinter.Scale*. The resulting spring movement profile will immediately be shown by the graph in the profile canvas. It is evident that for certain choices of parameter values, the displacement of the spring will exceed its normal interval [-1, 1]. As mentioned in the `widgets` module (chapter 10), this is allowed, but will lead to a warning message being printed, and in some cases, unnatural looking springs! (A spring with a negative length, for instance.)

The buttons “Start animation”, “Help” and “Exit”, starts the animation, displays the reference page (a *html* version of this section), and quits the application.

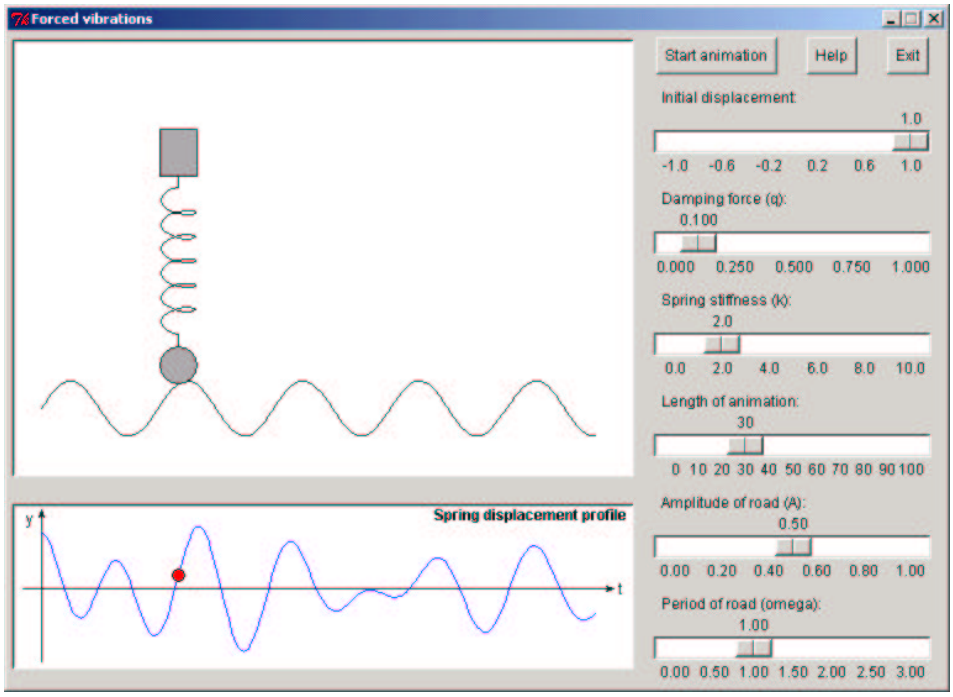


Figure 16.4: Snapshot of the application in action.

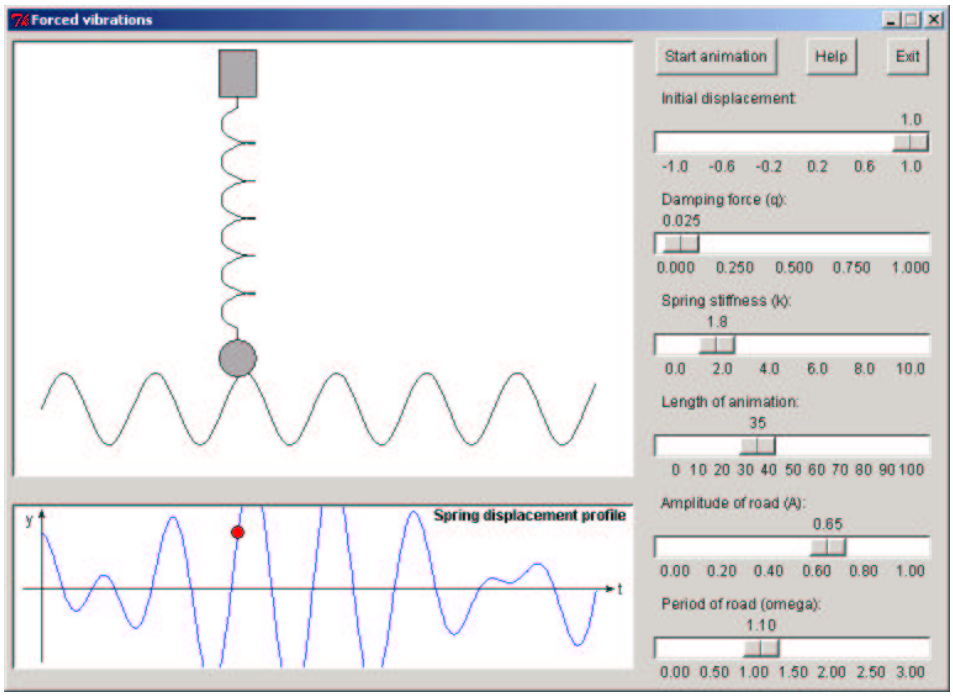


Figure 16.5: The effect of resonance is achieved with $q \sim 0$ and $\omega \sim \sqrt{k}$

16.5.5 Code

```
#!/bin/sh
""":"
exec python $0 ${1+"$@"}
"""

# Make sure the modules are found
import sys
sys.path[:0] = ['../']

# This will enable the program to be run from idle
import os
dirname,basename = os.path.split(sys.argv[0])
os.chdir(os.path.normpath(dirname))

import Tkinter
import Pmw
from modules import widgets
from modules import functions
from vibrations import system
from vibrations import basegui
from math import *


class Gui(basegui.BaseGui):
    """Handle interface and user interaction"""

    def __init__(self):
        # Path to the reference page
        self.refPage = '../doc/' + 'applications_vibrations_forced.html'

        # Initialise base class
        basegui.BaseGui.__init__(self, root)

        # Create the main canvas
        self.canvas = self.create_canvas(500, 350)
        self.canvas.grid(rowspan = 5, column = 0, padx = 2, pady = 2)

        # Create the buttons
        if os.name == 'nt':
            padx = 12
        else:
            padx = 0
        self.buttonBox = Pmw.ButtonBox(root, padx = padx)
        self.buttonBox.add('Start animation', command = self.animate_spring)
        self.buttonBox.add('Help', command = self.help)
        self.buttonBox.add('Exit', command = self.exit)
        self.buttonBox.grid(row = 0, column = 1, padx = 2, pady = 2)

        # Create the spring displacement profile canvas
        self.profileCanvas = self.create_canvas(500, 130)
        self.profileCanvas.grid(row = 5, rowspan = 2, padx = 2, pady = 2)
        self.draw_profileCanvas_stuff()

        # Create the scales
        list = (('Initial displacement:', -1, 1, 2, .4, .1, 1),
                ('Damping force (q):', 0.0, 1.0, 4, .25, .025, 0.1),
                ('Spring stiffness (k):', 0.0, 10.0, 3, 2, .1, 2.0),
```

```
            ('Length of animation:', 0, 100, 2, 10, 5, 30),
            ('Amplitude of road (A):', 0, 1, 3, .2, .05, .5),
            ('Period of road (omega):', 0, 3, 3, .5, .05, 1))
    self.scales = []
    for args in list:
       scale = self.create_scale(*args)
       scale.grid(row = len(self.scales)+1, column = 1, padx = 2)
       self.scales.append(scale)

    # Class ForcedVibrations handles the mathematical model of the system
    self.system = system.ForcedVibrations()

    # Create the spring
    self.spring = widgets.Spring(self.canvas, 25, 50, 120)


# Draw the spring at time t with given displacement
def draw_spring(self, t, displacement):
    self.canvas.delete('spring')

    # Calculate the spring's x- and top y-value
    y = self.road(t)
    cx,cy = self.find_canvas_values(((t,y),), base_y = 300)
    length = self.spring.get_total_length(displacement)
    self.spring.x = cx
    self.spring.y = cy - 30 - length

    # Draw box at the top of the spring
    y = self.spring.y
    x = self.spring.x
    kw = {'fill': 'DarkGray', 'tags': 'spring'}
    self.canvas.create_rectangle(x-15,y-38, x+15,y, **kw)

    # Draw spring
    self.spring.draw(displacement)

    # Draw wheel at the end of the spring
    x,y = self.spring.get_end_position(displacement)
    self.canvas.create_oval(x-15,cy-30, x+15,cy, **kw)


# Update all parameter values. Draw road, spring, and profile.
def new_parameter_value(self, dummy = 0):
    y0 = self.scales[0].get()
    q = self.scales[1].get()
    k = self.scales[2].get()
    t_max = self.scales[3].get()
    A = self.scales[4].get()
    omega = self.scales[5].get()

    self.system.new_parameter_values(y0, q, k, t_max, A, omega)

    self.draw_road()
    self.draw_spring(0, y0)
    self.draw_profile()


# Calculate and draw the corrugated road
def draw_road(self):
```

```
        A = self.system.A
        omega = self.system.omega
        u = self.system.u

        # road: f(u*t) = A*sin(omega*u*t)
        function = str(A) + '*sin('+str(omega) + '*' + str(u) + '*t)'
        self.road = functions.Function(function, 0, self.system.t_max,
                variable = 't',
                samples = 150)
        values = self.road.calculate_values()

        # Calculate the corresponding canvas values
        cvalues = self.find_canvas_values(values, base_y = 300)

        # Draw the road
        self.canvas.delete('road')
        self.canvas.create_line(cvalues, fill = 'Black', tags = 'road')
#----

root = Tkinter.Tk()
root.title('Forced vibrations')

if os.name in ('nt', 'posix'):
    root.option_readfile('fontsAndColors.txt')
else:
    Pmw.initialise(root, fontScheme = 'pmw1')

Gui()

root.resizable(0,0)
root.mainloop()
```

Chapter 17

Solution of nonlinear equations

17.1 Introduction

A frequent problem in numerical analysis is locating roots of equations, i.e. finding x such that $f(x) = 0$. This usually means solving a nonlinear equation. Several methods exist, but the bisection method, Newton's method and the secant method are probably the best known. All these methods have similar geometrical interpretations. This may be exploited by creating a common base class and a template file. We have already seen some examples of this strategy; in chapter 15 we looked at illustrations of five different sorting algorithms and how they all were created by means of two base classes and a template file. A similar method was used in chapter 16. We are therefore only going to look at one application this time; Newton's method. However, the base class and the template are created with several applications in mind.

17.2 Base GUI class

This class creates the graphical user interface, defines keyboard shortcuts and button bindings, and handles drawing of graphs and symbols. A derived class has to be created in the actual application file and this class needs some attributes and functions that `BaseGui` expects. These will be explained in the template section (17.3).

Several functions are available:

- `create_entryfield(label, initialvalue, width)` - Create and return a *Pmw.EntryField* with the given attributes.
- `create_checkbuttons()` - Create and return two checkbuttons. They are "Display line y=0" and "Manual zoom control". These should be useful in most applications of this type.
- `_create_checkbutton(text, var, command)` - A conveniance function used by `create_checkbuttons`.
- `create_buttonbox()` - Create a *Pmw.ButtonBox* with the following buttons: "Restart", "Next step", "Zoom in", "Zoom out", "Restore", "Help" and "Exit". Most of these are handled within this class except from "Restart" and "Next step". The button box is given the attribute name `self.buttonbox`.

- `create_message_box()` - Create a *Pmw.ScrolledText* widget. This widget is given the attribute name `self.messages` and is used to display messages concerning algorithm progress.

- `create_blt_graph()` - Create a *Blt.Graph* widget. This widget is given the attribute name `self.graph` and is the main widget for illustrating the algorithms.

- `insert_message(text)` - Insert the message `text` at the end of the scrolled text widget.

- `plot_function(function, x0, x1)` - Draw `function` as a graph $y = f(x)$ in the graph widget. `function` should be a `Function` object (see the `functions` module (chapter 7)). `x0` and `x1` are the range of x values to be displayed in the widget. This is not necessarily the same as the plotted range which is decided when the `Function` object is initialised.

- `create_y_line()` - Draw a line $y = 0$ in the graph widget.

- `draw_point(x, y, name)` - Draw a point (the image given by the attribute `self.redball`) at the given coordinates in the graph widget. Give the point the name `name`. Execute automatic zoom if the "Manual zoom control" checkbutton has been created and its value is off.

- `draw_line(x0,y0, x1,y1, text, name)` - Draw a stippled line in the graph widget from coordinates (`x0`, `y0`) to coordinates (`x1`, `y1`). Display the `text` just above or below (depending of the sign of `y1`) the end position. Give the line the name `name` and the text the name `name` + "txt".

- `_LMB_pressed()` - Find the graph coordinate value corresponding to the mouse cursor at the time of left mouse button clicks. Call the function `system.point_selected` (see the template section) with this value.

- `_zoom_in(pst = 0.9)` - Zoom in on the graph. The value of `pst` determines the amount.

- `_zoom_out(pst = 1.11)` - Zoom out of the graph. The value of `pst` determines the amount.

- `_restore()` - Restore the axes (or the zoom level) to its original state.

- `_toggle_axis()` - Display or hide the $y = 0$ graph widget line.

- `_toggle_zoom_mode()` - Change the zoom mode between manual and automatical. Disable the buttons "Zoom in", "Zoom out" and "Restore" if in automatical mode.

- `_bindings()` - Define keyboard bindings.

- `_help()` - Display reference page.

- `_exit()` - Quit application.

17.2.1 Code

```
#!/bin/sh
""":"
exec python $0 ${1+"$@"}
"""

# Make sure the modules are found
import sys
sys.path[:0] = ['../']

import Tkinter
import Pmw
import os
import thread

class BaseGui:
    """Base GUI class for the equation solvers"""

    def __init__(self, master):
        self.master = master
        self._bindings()

        self.redball = Tkinter.PhotoImage(file = 'redball.gif')

    # Create and return a Pmw.EntryField
    def create_entryfield(self, label, initialvalue, width):
        entryfield = Pmw.EntryField(self.master,
                entry_background = 'White',
                entry_width = width,
                labelpos = 'w',
                label_text = label,
                value = initialvalue)
        return entryfield

    # Create and return two checkbuttons and
    def create_checkbuttons(self):
        self.axisVar = Tkinter.IntVar()
        self.axisVar.set('1')
        self.zoomVar = Tkinter.IntVar()
        self.zoomVar.set('1')
        cb1 = self._create_checkbutton('Display line y=0', self.axisVar,
                self._toggle_axis)
        cb2 = self._create_checkbutton('Manual zoom control', self.zoomVar,
                self._toggle_zoom_mode)
        return cb1,cb2

    # Create and return a Tkinter.Checkbutton
    def _create_checkbutton(self, text, var, command):
        checkbutton = Tkinter.Checkbutton(self.master,
                command = command,
                text = text,
                var = var)
        return checkbutton

    # Create a Pmw.ButtonBox and some buttons
    def create_buttonbox(self):
        if os.name == 'nt':
            padx = 2
```

```
            ipadx = 2
        else:
            padx = 0
            ipadx = 3
        self.buttonbox = Pmw.ButtonBox(self.master, padx = padx)
        self.buttonbox.add('Restart', command = self.system.restart,
                padx = ipadx)
        self.buttonbox.add('Next step', command = self.system.flow.next_step,
                padx = ipadx)
        self.buttonbox.add('Zoom in', command = self._zoom_in, padx = ipadx)
        self.buttonbox.add('Zoom out', command = self._zoom_out, padx = ipadx)
        self.buttonbox.add('Restore', command = self._restore, padx = ipadx)
        self.buttonbox.add('Help', command = self._help, padx = ipadx)
        self.buttonbox.add('Exit', command = self._exit, padx = ipadx)
        self.buttonbox.alignbuttons()

    # Create a Pmw.ScrolledText widget
    def create_message_box(self):
        self.messages = Pmw.ScrolledText(self.master,
                labelpos = 'nw',
                label_text = 'Messages:',
                text_background = 'White',
                text_height = 6,
                text_state = 'disabled',
                text_width = 48,
                vscrollmode = 'static')

    # Create a BLT.Graph widget
    def create_blt_graph(self):
        self.graph = Pmw.Blt.Graph(self.master,
                height = 400,
                width = 600)
        self.graph.xaxis_configure(min = 0.0, max = 1.0)
        self.create_y_line()
        self.graph.bind('<Button-1>', self._LMB_pressed)

    # Insert text in the message box
    def insert_message(self, text):
        self.messages.configure(text_state = 'normal')
        self.messages.insert('end', text + '\n')
        self.messages.see('end')
        self.messages.configure(text_state = 'disabled')

    # Draw a function in the blt graph window
    def plot_function(self, function, x0, x1):
        # Clear graph window
        for element in self.graph.element_names():
            self.graph.element_delete(element)
        self.create_y_line()

        # Draw the graph
        xdata = []
        ydata = []
        for x,y in function.calculate_values():
            xdata.append(x)
            ydata.append(y)
        self.graph.line_create('function', xdata = tuple(xdata), label = '',
                ydata = tuple(ydata), symbol = '', smooth = 'natural')
```

```
        # Resize graph axis
        y0 = function.min((x0,x1))
        y1 = function.max((x0,x1))
        if y0 >= 0:
            y0 = -y0/10.0
        self.origx0,self.origx1 = x0,x1
        self.origy0,self.origy1 = y0,y1
        self.graph.xaxis_configure(min = x0, max = x1)
        self.graph.yaxis_configure(min = y0, max = y1)

    # Draw the y=0 line in the graph widget
    def create_y_line(self):
        kw = {'xdata': (-10000,10000), 'ydata': (0,0), 'symbol': '',
              'label': '', 'color': 'black', 'linewidth': 1}
        self.graph.line_create('y=0', **kw)
        self._toggle_axis()

    # Draw point at (x,y)
    def draw_point(self, x, y, name):
        xrange = self.graph.axis_limits('x')[1] - self.graph.axis_limits('x')[0]
        self.graph.marker_create('image', name = name, image = self.redball,
                coords = (x+xrange/200.0,y), under = 1)

        # Zoom automatically if not in manual mode
        yrange = self.graph.axis_limits('y')[1] - self.graph.axis_limits('y')[0]
        try:
            if (self.zoomVar.get() == 0 and abs(y/yrange) < 0.5):
                self._zoom_in(0.5 + abs(y/yrange))
        except:
            pass

    # Draw a line with given text
    def draw_line(self, x0,y0, x1,y1, text, name):
        yrange = self.graph.axis_limits('y')[1] - self.graph.axis_limits('y')[0]
        self.graph.marker_create('line', name = name,
                coords = (x0,y0,x1,y1), dashes = (5,), under = 1)
        if y0 < 0:
            yrange = -yrange
        self.graph.marker_create('text', name = name + 'txt',
                coords = (x1,-yrange/20.0), text = text, under = 1)

    # Return the graph coordinate value corresponding to the mouse cursor
    def _LMB_pressed(self, event = None):
        if not (self.system.selectPointMode and
                self.graph.inside(event.x, event.y)):
            return
        x = self.graph.invtransform(event.x, event.y)[0]
        self.system.point_selected(x)

    # Decrease axis' range
    def _zoom_in(self, pst = 0.9):
        xrange = self.graph.axis_limits('x')[1] - self.graph.axis_limits('x')[0]
        min = self.system.x - pst*xrange/2.0
        max = self.system.x + pst*xrange/2.0
        self.graph.xaxis_configure(min = min, max = max)
        y0,y1 = self.graph.axis_limits('y')
        self.graph.yaxis_configure(min = pst*y0, max = pst*y1)

    # Increase axis' range
```

```
def _zoom_out(self, pst=1.11):
    xrange = self.graph.axis_limits('x')[1] - self.graph.axis_limits('x')[0]
    min = self.system.x - pst*xrange/2.0
    max = self.system.x + pst*xrange/2.0
    self.graph.xaxis_configure(min = min, max = max)
    y0,y1 = self.graph.axis_limits('y')
    min = y0 - (pst - 1.0)*(y1 - y0)/2.0
    max = y1 + (pst - 1.0)*(y1 - y0)/2.0
    self.graph.yaxis_configure(min = min, max = max)

# Restore axis to original min/max
def _restore(self):
    x0,x1 = self.origx0, self.origx1
    y0,y1 = self.origy0, self.origy1
    self.graph.xaxis_configure(min = x0, max = x1)
    self.graph.yaxis_configure(min = y0, max = y1)

# Display/hide line y=0
def _toggle_axis(self):
    y0,y1 = self.graph.axis_limits('y')
    if self.axisVar.get() == 0:
        value=1
    else:
        value=0
    self.graph.element_configure('y=0', hide = value)

    # Force y-axis range to stay the same
    self.graph.yaxis_configure(min = y0, max = y1)

# Change zoom control
def _toggle_zoom_mode(self):
    if self.zoomVar.get() == 1:
        state='normal'
    else:
        state = 'disabled'
    for component in ('Zoom in', 'Zoom out', 'Restore'):
        self.buttonbox.component(component).configure(state = state)

# Create keyboard bindings
def _bindings(self):
    self.master.bind('<q>', self._exit)
    self.master.bind('<h>', self._help)

# Display the reference page
def _help(self, event = None):
    if os.name == 'nt':
        app = os.path.normpath(self.refPage)
    else:
        app = 'netscape ' + os.path.normpath(self.refPage) + '&'
    thread.start_new_thread(os.system, (app,))

# Quit application
def _exit(self, event = None):
    self.master.destroy()
```

17.3 Template

17.3.1 Gui

This class is a subclass of the `BaseGui` class described in section 17.2. Besides defining the graphical user interface, not much should be needed here. The base class expects these attributes:

- `self.refPage` - Path to the *html* reference page. This will be displayed whenever the "Help" button is pressed.
- `self.system` - An object of the `System` class described in the next section.

Create widgets with the functions from the base class. A suitable selection and positioning is suggested in the Code section (17.3.3). Note that some widgets created by this method will automatically be given attribute names.

17.3.2 System

This is were most of the work is done. The attribute `self.flow` is mandatory and should be an object of type `ProgramFlow` (see the `programflow` module (chapter 9)). This class will handle pauses and restarts of the algorithm. The `self.selectPoint-Mode` is also needed as it lets the `BaseGui` class know if a selection in the graph widget is allowed. Some other attributes are suggested, but none of them are required.

At least three functions are needed:

- One that handles the actual algorithm. It is called `newtons_method` in the Newton's method application. This must be given as an argument to the `ProgramFlow` class when the `self.flow` object is created.
- `restart` - will be called from the `BaseGui` class whenever the "Restart" button is pressed.
- `point_selected(x)` - will be called if the `self.selectPointMode` attribute is `True` and the user clicked the left mouse button inside the graph widget. The `x` value is the x graph coordinate of the mouse cursor.

17.3.3 Code

```
#!/bin/sh
""":"
exec python $0 ${1+"$@"}
"""

# Make sure the modules are found
import sys
sys.path[:0] = ['../']

# This will enable the program to be run from idle
import os
dirname,basename = os.path.split(sys.argv[0])
os.chdir(os.path.normpath(dirname))

import Tkinter
```

```
import Pmw
from equationsolvers import baseclasses
from modules import programflow
from modules import functions


class Gui(baseclasses.BaseGui):
    """Handle interface and user interaction"""

    def __init__(self):
        # Path to the reference page
        self.refPage = '../doc/' + 'applications_equationsolvers_****.html'

        # Initialise base class
        baseclasses.BaseGui.__init__(self, root)

        # Let the System class take care of the algorithm
        self.system = System(self)

        # Create widgets
        self.functionEF = self.create_entryfield('Function: ', 'sin(x)', 22)
        self.minXEF = self.create_entryfield('min_x', '0.0', 8)
        self.maxXEF = self.create_entryfield('max_x', '6.28', 8)
        cb1,cb2 = self.create_checkbuttons()
        self.create_buttonbox()
        self.create_message_box()
        self.create_blt_graph()

        # Position widgets
        root.grid_rowconfigure(0, minsize = 5)
        self.functionEF.grid(row = 1, columnspan = 2, column = 0, padx = 5)
        self.minXEF.grid(row = 1, column = 1, padx = 5)
        self.maxXEF.grid(row = 2, column = 1, padx = 5)
        cb1.grid(row = 1, column = 2, sticky = 'w', padx = 5)
        cb2.grid(row = 2, column = 2, sticky = 'w', padx = 5)
        root.grid_rowconfigure(3, minsize = 5)
        self.buttonbox.grid(row = 4, columnspan = 3, padx = 5)
        self.messages.grid(row = 5, columnspan = 3, padx = 5)
        self.graph.grid(row = 6, columnspan = 3, padx = 5, pady = 5)

        Pmw.alignlabels((self.minXEF, self.maxXEF))



class System:
    """Handle system specific calculations"""

    def __init__(self, gui):
        self.gui = gui
        self.flow = programflow.ProgramFlow(root, self._method)
self.selectPointMode = 0

        # These attributes are usually needed
        self.function = None    # The function (to be used in the algorithm)
        self.x0 = 0             # The definition set of the function [x0, x1]
        self.x1 = 0
        self.x = 0              # Current solution

    # This is the actual algorithm
```

```
    def _method(self):
        pass

    # Restart the algorithm
    def restart(self, event = None):
        self.flow.restart()

    # The user has just selected a point
    def point_selected(self, x):
        pass

#----

root = Tkinter.Tk()
root.title('**** method')

if os.name in ('nt', 'posix'):
    root.option_readfile('fontsAndColors.txt')
else:
    Pmw.initialise(root, fontScheme = 'pmw1')

gui = Gui()
root.resizable(0,0)
root.mainloop()
```

17.4 Newton's method

17.4.1 The algorithm

With a quadratic convergence rate, Newton's method is significantly faster than the bisection and secant methods. However, convergence is not guaranteed, and the success of the method relies heavily on the initial estimate, but once the estimates produced by the method are close to the root, usually only a few iterations are needed.

Let f be the sufficiently smooth function whose zero we want to determine. If r is a root of f and x_0 an approximation to r, then by Taylor's theorem:

$$0 = f(r) = f(x_0 + h) = f(x) + hf'(x) + \ldots, \quad h = r - x_0.$$

We may ignore higher order terms when h is sufficiently small, i.e. x_0 is close to r. Rearranging the equation leads to

$$h = -\frac{f(x)}{f'(x)},$$

which suggests the iterative method:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}, \quad n = 0, 1, 2, \ldots.$$

Further details concerning Newton's method can be found in [12].

17.4.2 The illustration

The first step in the illustration is to draw the graph $y = f(x)$. The "Next step" button executes every step of the algorithm except for the initial estimate selection described

below. To be able to zoom outwards, the graph is drawn on five times the x range given by the `min_x` and `max_x` entry fields. The initial displayed range is, however, set to [`min_x`, `max_x`]. When the graph is drawn, we need to select an initial estimate, or starting point. This is done by clicking the left mouse button while the mouse cursor is over the desirable x position in the graph widget. Now the iterative method starts and these two steps are repeated:

- Calculate the tangent of $f(x)$ at the current estimate (x_0). Show this as a stippled line and call the the intersection of the line and the x axis for x_1.
- Delete tangent line and use x_1 as the new estimate (rename this as x_0). Calculate the function value of the new estimate.

17.4.3 The user interface

The following widgets are used:

- Four *Pmw.EntryFields* used to input the function $f(x)$ and its derivative $f'(x)$ as text strings, and the minimum and maximum value of x. Remember to use Python type mathematical syntax and that x has to be the independent variable.
- Two *Tkinter.Checkbuttons* used to display or hide the $y = 0$ line and to change between manual and automatic zoom mode.
- A row of buttons (a *Pmw.ButtonBox*):
  - "Restart" - Start the algorithm from scratch.
  - "Next step" - Execute the next step of the algorithm.
  - "Zoom in" - Zoom in on the graph.
  - "Zoom out" - Zoom out of the graph.
  - "Restore" - Restore zoom level to initial value.
  - "Help" - Display the reference page (a *html* version of this section).
  - "Exit" - Quit application.
- A *Pmw.ScrolledText* widget to display messages.
- A *Blt.Graph* widget used to draw the graphical interpretation.

17.4.4 Code

```
#!/bin/sh
""":"
exec python $0 ${1+"$@"}
"""

# Make sure the modules are found
import sys
sys.path[:0] = ['../']

# This will enable the program to be run from idle
import os
dirname,basename = os.path.split(sys.argv[0])
```

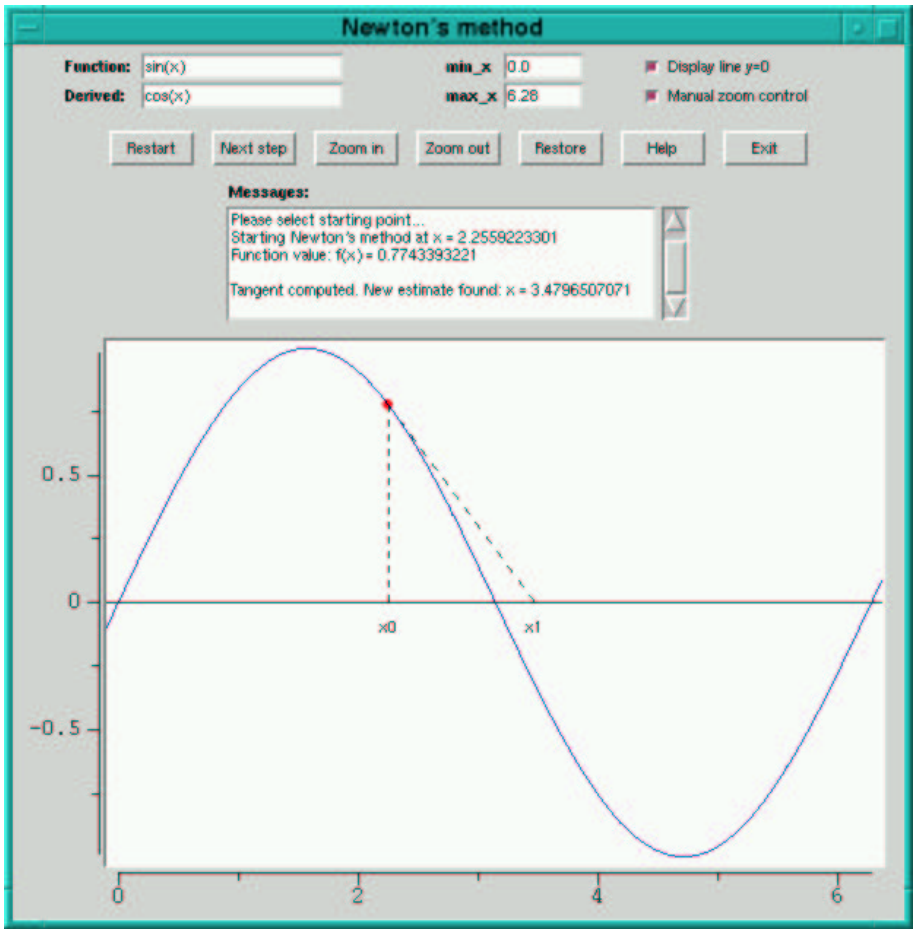


Figure 17.1: A graphical interpretation of Newton's method.

```
os.chdir(os.path.normpath(dirname))

import Tkinter
import Pmw
from equationsolvers import baseclasses
from modules import programflow
from modules import functions


class Gui(baseclasses.BaseGui):
    """Handle interface and user interaction"""

    def __init__(self):
        # Path to the reference page
        self.refPage = '../doc/' + 'applications_equationsolvers_newton.html'

        # Initialise base class
        baseclasses.BaseGui.__init__(self, root)

        # Let the System class take care of the algorithm
        self.system = System(self)

        # Create widgets
        self.functionEF = self.create_entryfield('Function: ', 'sin(x)', 22)
        self.derivedEF = self.create_entryfield('Derived: ', 'cos(x)', 22)
        self.minXEF = self.create_entryfield('min_x', '0.0', 8)
        self.maxXEF = self.create_entryfield('max_x', '6.28', 8)
        cb1,cb2 = self.create_checkbuttons()
        self.create_buttonbox()
        self.create_message_box()
        self.create_blt_graph()

        # Position widgets
        root.grid_rowconfigure(0, minsize = 5)
        self.functionEF.grid(row = 1, column = 0, padx = 5)
        self.derivedEF.grid(row = 2, column = 0, padx = 5)
        self.minXEF.grid(row = 1, column = 1, padx = 5)
        self.maxXEF.grid(row = 2, column = 1, padx = 5)
        cb1.grid(row = 1, column = 2, sticky = 'w', padx = 5)
        cb2.grid(row = 2, column = 2, sticky = 'w', padx = 5)
        root.grid_rowconfigure(3, minsize = 5)
        self.buttonbox.grid(row = 4, columnspan = 3, padx = 5)
        self.messages.grid(row = 5, columnspan = 3, padx = 5)
        self.graph.grid(row = 6, columnspan = 3, padx = 5, pady = 5)

        Pmw.alignlabels((self.functionEF, self.derivedEF))
        Pmw.alignlabels((self.minXEF, self.maxXEF))



class System:
    """Handle system specific calculations"""

    def __init__(self, gui):
        self.gui = gui
        self.flow = programflow.ProgramFlow(root, self.newtons_method)
        self.selectPointMode = 0

        self.function = None    # The function (to be used in the algorithm)
```

```
        self.derived = None     # The first derivative of the function
        self.x0 = 0             # The definition set of the function [x0, x1]
        self.x1 = 0
        self.x = 0              # Current solution

    # This is the actual algorithm
    def newtons_method(self):
        # Start by plotting the graph
        text = 'Plotting function\nPlease select starting point...'
        self.gui.insert_message(text)
        self.new_function()
        self.gui.plot_function(self.function, self.x0, self.x1)

        # Wait while the user selects a starting point
        self.selectPointMode = 1
        self.gui.buttonbox.component('Next step').configure(state = 'disabled')
        self.flow.wait_or_next_step()

        while self.flow.started:
            # Compute the tangent and find new estimate
            x = self.x = self.compute_tangent()
            text = 'Tangent computed. New estimate found: ' +\
                   'x = %(x)10.10f' %vars()
            self.gui.insert_message(text)
            self.flow.wait_or_next_step()

            # Calculate new function value
            y = self.function(x)
            text = 'Function value: f(x) = %(y)10.10f\n' %vars()
            self.gui.insert_message(text)
            self.gui.graph.marker_delete('dot', 'line1', 'line1txt',
                    'line2', 'line2txt')
            self.gui.draw_point(x, y, 'dot')
            self.gui.draw_line(x,y, x,0, 'x0', 'line1')

            self.flow.wait_or_next_step()

    # Restart the algorithm
    def restart(self, event = None):
        self.gui.graph.marker_delete('dot', 'line1', 'line1txt',
                'line2', 'line2txt')
        self.gui.messages.clear()
        self.gui.buttonbox.component('Next step').configure(state = 'normal')
        self.flow.restart()

    # Define the function by the current entry fields' values
    def new_function(self):
        self.x0 = float(self.gui.minXEF.get())
        self.x1 = float(self.gui.maxXEF.get())
        self.x = (self.x1 - self.x0)/2.0

        # Calculate 5 times the given x range.
        # (Used to zoom outwards)
        xmin = self.x0 - 5*(self.x1 - self.x0)
        xmax = self.x1 + 5*(self.x1 - self.x0)

        # Create Function objects to manage calculations
        # of the function and its first derivative
        func = self.gui.functionEF.get()
```

```
        self.function = functions.Function(func, xmin, xmax)
        func = self.gui.derivedEF.get()
        self.derived = functions.Function(func, xmin, xmax)

    # The user has just selected a starting point
    def point_selected(self, x):
        self.x = x
        text = 'Starting Newton\'s method at x = %(x)10.10f' %vars()
        self.gui.insert_message(text)

        y = self.function(x)
        self.gui.draw_point(x, y, 'dot')
        self.gui.draw_line(x,y, x,0, 'x0', 'line1')
        text = 'Function value: f(x) = %(y)10.10f\n' %vars()
        self.gui.insert_message(text)

        self.selectPointMode = 0
        self.gui.buttonbox.component('Next step').configure(state = 'normal')

    # Compute tangent and find new estimate
    def compute_tangent(self):
        x0 = self.x
        y0 = self.function(self.x)
        dy = self.derived(x0)
        x1 = x0 - y0/dy
        self.gui.draw_line(x0,y0, x1,0, 'x1', 'line2')
        return x1
#----

root = Tkinter.Tk()
root.title('Newton\'s method')

if os.name in ('nt', 'posix'):
    root.option_readfile('fontsAndColors.txt')
else:
    Pmw.initialise(root, fontScheme = 'pmw1')

gui = Gui()
root.resizable(0,0)
root.mainloop()
```

Part IV

Concluding words

Chapter 18

Summary and conclusion

We have seen that Python and the toolkits Tkinter, Pmw and Blt are powerful building blocks for programming visual illustrations. The ability to extend that toolkit with specialised modules make programming on a high abstract level possible. Python's simple and elegant syntax support the effort to write highly readable code.

18.1 Creating modules

Some of the modules presented here were discovered by creating sketches of applications and finding what they had in common (`BasePrefs`, `calculate_geometry`, `create_path`, `ProgramFlow` to mention a few). Others were created on the basis of general Python experience; when the possibilities and restrictions of the language are known, some ideas instantly spring to mind. The Pmw widgets (`MultiListBox` and `ProgressBarDialog`) and the `SmoothRectangle` are examples of this.

18.1.1 Generality versus specialisation

A legitimate question is whether a module should be as general as possible or if it should mainly be made to serve the current problem. I am all for the general approach as a starting point, but it requires substantially more work. The modules in part II are examples of both general and very specialised bits of code. The Pmw widgets and the `Physical2CanvasSystem` class should be at the general end of the scale, while the classes `BasePrefs` and `ProgramFlow` populates the other end. The other functions and classes are probably distributed pretty evenly in between.

18.2 The applications

Some remarks about the effects of the applications.

- **Viscous flow between parallel plates** - The simple way to change the basic properties of the flow, by pressing the symbols in the canvas, is the best feature of this illustration. It lacks, however, the ability to change the physical parameters directly by dragging scales and seeing the results immediately, as is the case for the `vibrations` applications. Scales are brilliant, but they are often difficult to use because of the great space they require and the missing option to change the label position.

- **The matrix product** - The effect of this illustration is greatest if the students could try different matrix sizes and play around with the program. This makes it a good candidate for a web application or a downloadable version the students could execute locally.

- **The Runge-Kutta method** - This is more of a useful standalone tool than an active teaching tool. Nevertheless, it works fine and looks good. The ability to plot only selected solutions is a simple but nice feature. Adjusting entry fields values in the current system is a little bothersome and should perhaps be fixed.

- **Sorting algorithms** - These turned out very well. Speaking out of personal experience, I would never have been able to remember exactly how the different algorithms work if it hadn't been for these illustrations. This is a very fine example on the effect of dynamic illustrations. In [10], for example, seven pages are used to visualise the merge-sort and quick-sort algorithms. That isn't a very practical way to teach it in a classroom.

- **Vibrations in mechanical systems** - This is perhaps the application I am most pleased about. The immediate results of changing the physical parameters by dragging the scales are very appealing and the spring animations look good. A possible extension could be to introduce the mass of the car or the horizontal velocity as adjustable parameters.

- **Solution of nonlinear equations** - The main effect here is basically to show how fast Newton's method converges once the estimates get closer to the root. The overall GUI is perhaps the least visually pleasing of the applications.

18.3 Running GUIs on different platforms

Although Python and Tkinter/Pmw/Blt are platform independent, some individual adjustments usually have to be applied. This is especially the case with fonts and sizes. The spacing between buttons in the *Pmw.ButtonBox* widget, for instance, is quite different on Windows 2000 and some Linux/Unix versions. This complicates matters a little if 100% finished applications are the goal. No big efforts have been made in the applications presented here to achieve this, but the method of testing os type by using the `os.name` attribute has been used in several places.

18.4 Results

One of the purposes of this thesis was to explore the suitability of Python for the task of creating visual illustrations. My experience is that using Python with a set of specialised tools leads to short, readable code with more opportunities than restrictions. With a high level of code re-use, many new applications can be very swiftly coded. If we take into consideration that it only takes a couple of hours to learn to use Python productively, it should be very well suited, even for non-programming-experts.

All of the applications in part III are taken from subjects that are teached at an undergraduate level. It is perhaps at this level the effects of visual imaging are most effectful, or maybe most needed. In my opinion, most of the applications could be

taken into the classroom and used directly, without further improvements, but more importantly it should inspire teachers to create their own illustrations.

Chapter 19

Future work

If a solid toolkit for visualisation of dynamic processes in Python are to be built, a number of different tasks remain outstanding. Both more general and more specialised modules could be useful. Several new canvas widgets, like the `Spring` or the `SmoothRectangle`, should come in handy in applications in physics, and composite widget types (like Pmw widgets) are interesting in almost any type of GUI application. It would also be nice to have some more general environment description modules; classes handling, for example, gravitational objects, magnetic fields or forces due to friction.

A venture into the world of three dimensions is beyond the scope of this text, but if a simple interface could be created, it might prove a valuable addition to the toolbox.

Another direction this subject could go is the creation of an all-GUI illustration creator. This would give non-programmers a chance to create illustrations by means of manipulating symbols and writing algorithm details in an easy-to-use script language. It will be hard to create very complicated illustrations with such a tool, but it might function as an electronic blackboard, giving the teacher something with which to create simple on-the-fly illustrations.

Appendix A

Templates, demonstrations and miscellaneous

A.1 Application template

Almost all my applications contain some common elements: the python header, some import statements, initialisation of a root window, some font and colour definitions. In addition, these three classes are usually needed: `Gui` (the interface and user interaction), `System` (calculations and algorithms), and `Prefs` (preference values). The template application file can be used as a shell, or a building block, when new applications are started.

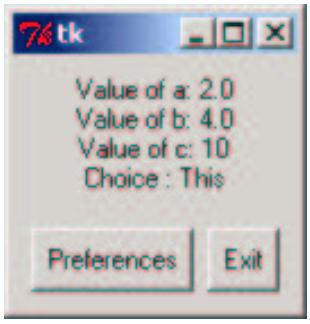
```
#!/bin/sh
""":"
exec python $0 ${1+"$@"}
"""

# Make sure the modules are found
import sys
sys.path[:0] = ['../']

# This will enable the program to be run from idle
import os
dirname,basename = os.path.split(sys.argv[0])
os.chdir(os.path.normpath(dirname))

import Tkinter
import Pmw
import thread
from modules import misc
from modules import dialogs


class Gui:
    """Handle interface and user interaction"""

    def __init__(self):
        pass

    def bindings(self):
        root.bind('<q>', self.exit)
```

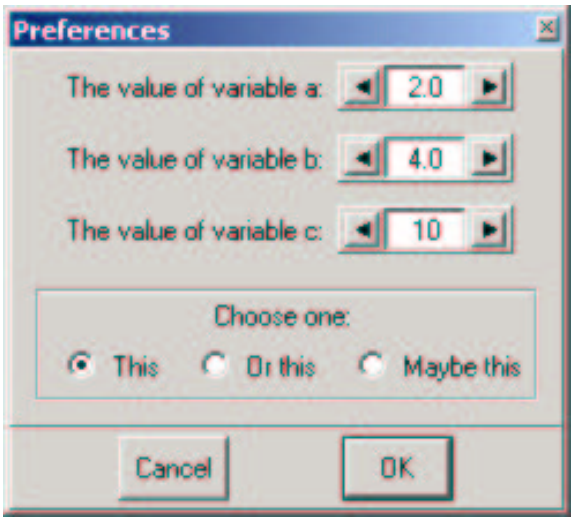
```
        root.bind('<h>', self.help)

    def help(self, event = None):
        refPage = 'applications_****.html'
        if os.name == 'nt':
            app = os.path.normpath('../doc/' + refPage)
        else:
            app = 'netscape ' + os.path.normpath('../doc/' + refPage) + '&'
        thread.start_new_thread(os.system, (app,))

    def exit(self, event = None):
        root.destroy()


class System:
    """Handle system specific calculations"""

    def __init__(self):
        pass


class Prefs(dialogs.BasePrefs):
    """Create standard preference dialog"""


    def __init__(self):
        dialogs.BasePrefs.__init__(self)

    def launch_window(self):
        self.create_dialog(root, width = 240, height = 390)
        list = ()
        self.counters = self.create_counters(list)

    def close_window(self, button):
        if button == 'OK':
            pass
        self.dialog.destroy()
#----

root = Tkinter.Tk()
root.title('*Template*')

if os.name in ('nt', 'posix'):
    root.option_readfile('fontsAndColors.txt')
else:
    Pmw.initialise(root, fontScheme = 'pmw1')

gui = Gui()
root.resizable(0,0)
root.mainloop()
```

A.2 BasePrefs demonstration

This is a demonstration of how to use the `BasePrefs` class found in the `dialogs` module (chapter 6).

```
#!/bin/sh
""":"
```


tk
Value of a: 2.0
Value of b: 4.0
Value of c: 10
Choice : This
Preferences
Exit


Preferences
The value of variable a:
2.0
The value of variable b:
4.0
The value of variable c:
10
Choose one:
This
Or this
Maybe this
Cancel
OK

```
exec python $0 ${1+"$@"}
"""

# Make sure the modules are found
import sys
sys.path[:0] = ['../']

import Tkinter
import Pmw
import os
from modules import dialogs

class Gui:
    def __init__(self):
        self.balloon = Pmw.Balloon(root)
        self.prefs = Prefs(self)

        self.label = Tkinter.Label(root)
        self.label.pack(padx = 5, pady = 5)
        self.display_values()

        buttonbox = Pmw.ButtonBox(root)
        buttonbox.add('Preferences', command = self.launch_preferences)
        buttonbox.add('Exit', command = self.exit)
        buttonbox.pack(padx = 5, pady = 5)

    def display_values(self):
        text = 'Value of a: ' + str(self.prefs.a) + '\n' +\
               'Value of b: ' + str(self.prefs.b) + '\n' +\
               'Value of c: ' + str(self.prefs.c) + '\n' +\
               'Choice : ' + self.prefs.choice
        self.label.configure(text = text)

    def launch_preferences(self):
        self.prefs.launch_window()

    def exit(self):
        root.destroy()
#----

class Prefs(dialogs.BasePrefs):
    def __init__(self, gui):
        self.gui = gui

        # Initialise base class
        dialogs.BasePrefs.__init__(self, gui.balloon)

        # Create default values
        self.a = 2.0
        self.b = 4.0
        self.c = 10
        self.choice = 'This'

    def launch_window(self):
        if os.name == 'nt':
            width = 240
            height = 200
        else:
            width = 350
```

```
            height = 230
        self.create_dialog(root, width = width, height = height)

        # All counters may be specified in this list
        # (label, variable, type, min, max, balloon_text)
        list = (('The value of variable a: ', self.a, 'real', 0.0, 5.0,
                 'a is a real number (0.0 - 5.0)'),
                ('The value of variable b: ', self.b, 'real', 0.0, 10.0,
                 'b is a real number (0.0 - 10.0)'),
                ('The value of variable c: ', self.c, 'integer', 1, 25,
                 'c is an integer (1 - 25)'))
        self.counters = self.create_counters(list)

        # Other widget types may be create as this radioselect
        # (label, buttons, pady)
        args = ('Choose one: ', ('This', 'Or this', 'Maybe this'), 10)
        self.radioSelect = self.create_radioselect(*args)
        self.radioSelect.invoke(self.choice)

    # This function is called whenever a button in the dialog is pressed
    def close_window(self, button):
        if button == 'OK':
            self.a = float(self.counters[0].get())
            self.b = float(self.counters[1].get())
            self.c = int(self.counters[2].get())
            self.choice = self.radioSelect.getcurselection()
            self.gui.display_values()
        self.dialog.destroy()
#----

root = Tkinter.Tk()
Gui()
root.mainloop()
```

A.3 Function demonstration

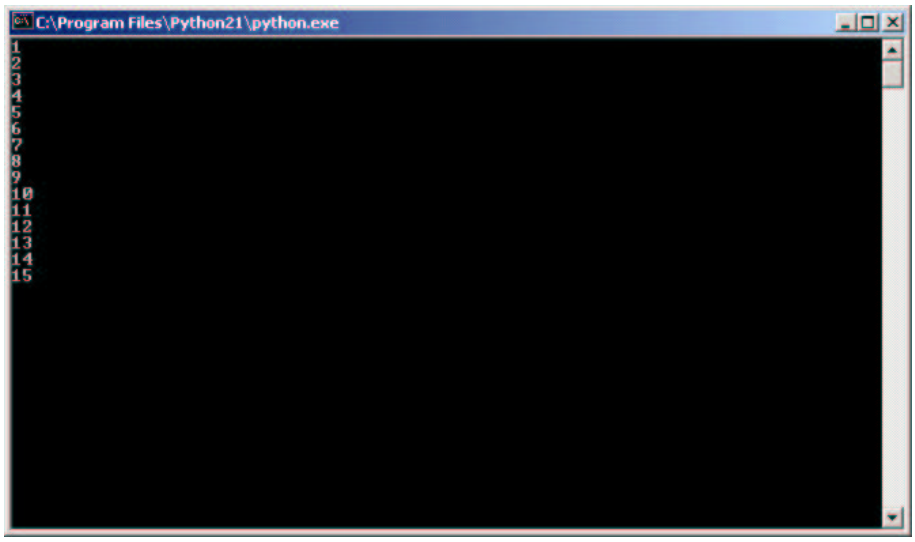
This is a demonstration of how to use the `Function` class found in the `functions` module (chapter 7).

```
#!/bin/sh
""":"
exec python $0 ${1+"$@"}
"""

# Make sure the modules are found
import sys
sys.path[:0] = ['../']

import time
from modules import functions
from math import *

function = functions.Function('sin(x)', 0, 2*pi, samples = 20)
for x,y in function.calculate_values():
    print 'x: %(x)6.6f   f(x): %(y)6.6f' %vars()
print 'min:', function.min()
print 'max:', function.max(interval = [0, pi/4])
```

```
time.sleep(2)
```

A.4 ProgramFlow demonstration

This is a demonstration of how to use the `ProgramFlow` class found in the `programflow` module (chapter 9).



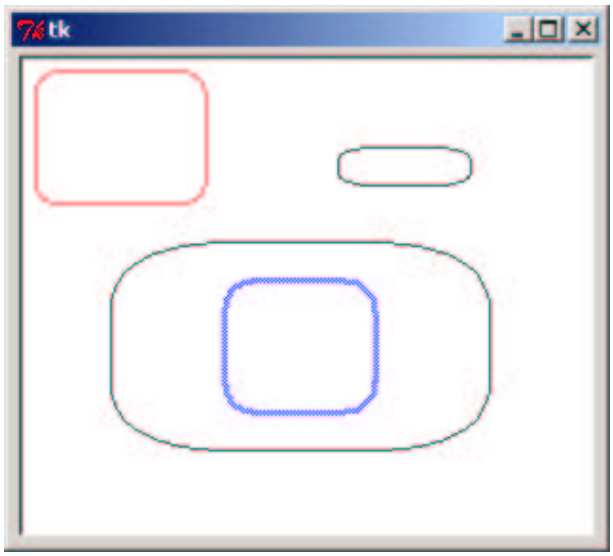

```
#!/bin/sh
"""":"
exec python $0 ${1+"$@"}
"""

# Make sure the modules are found
import sys
sys.path[:0] = ['../']

import Tkinter
import time
from modules import programflow

class Gui:
    def __init__(self):
        self.flow = programflow.ProgramFlow(root, self.algorithm)

        self.buttons = []
        list = (('Animation', self.animation, 0, 0),
                ('Step by step', self.next_step, 0, 1),
```

```
                ('Restart', self.restart, 1, 0),
                ('Pause/stop', self.stop_animation, 1, 1))
        for text,cmd,row,col in list:
            button = Tkinter.Button(text = text, command = cmd)
            button.grid(row = row, column = col, padx = 5, pady = 5)
            self.buttons.append(button)
        self.buttons[3].configure(state = 'disabled')

    def animation(self):
        self.buttons[3].configure(state = 'normal')
        self.flow.start_animation()

    def next_step(self):
        self.buttons[3].configure(state = 'disabled')
        self.flow.next_step()

    def restart(self):
        time.sleep(.2)
        self.flow.restart()

    def stop_animation(self):
        self.buttons[3].configure(state = 'disabled')
        self.flow.stop_animation()

    def algorithm(self):
        for i in range(20):
            print i + 1
            self.flow.wait_or_next_step(seconds = .2)
        self.flow.animation = 0
        self.flow.started = 0
        self.buttons[3].configure(state = 'disabled')
#----

root = Tkinter.Tk()
Gui()
root.mainloop()
```

A.5 SmoothRectangle demonstration

This is a demonstration of how to use the `SmoothRectangle` class found in the `widgets` module (chapter 10).

```
#!/bin/sh
""":"
exec python $0 ${1+"$@"}
"""

# Make sure the modules are found
import sys
sys.path[:0] = ['../']

import Tkinter
from modules import widgets

root = Tkinter.Tk()

canvas = Tkinter.Canvas(root,
        background = 'White',
```

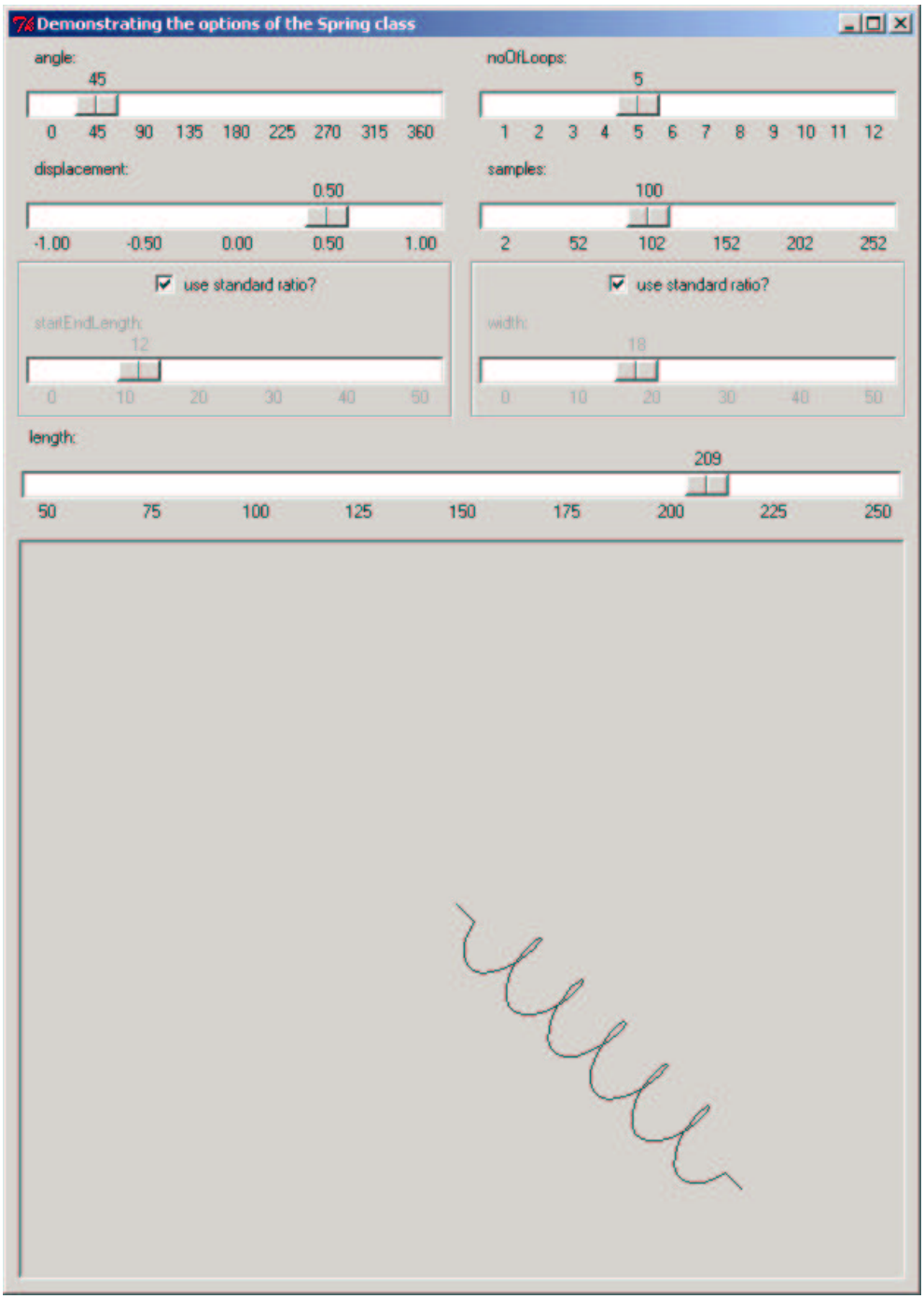


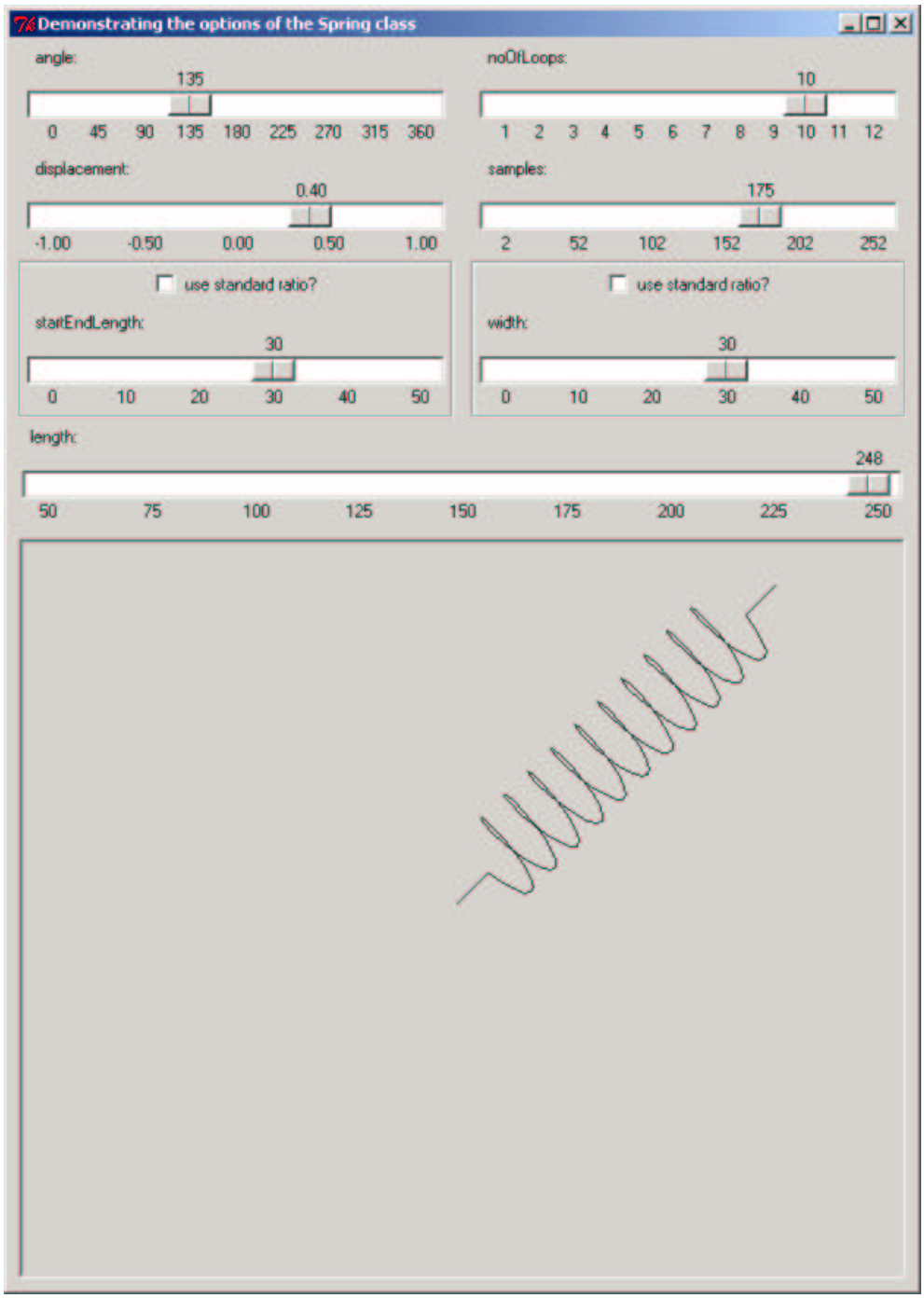

```
        borderwidth = 2,
        height = 250,
        relief = 'sunken',
        width = 300)
canvas.pack(padx = 2, pady = 2)

widgets.SmoothRectangle(canvas, 50,100, 250,210).draw()
widgets.SmoothRectangle(canvas, 10,10, 100,80, r = 20, outline = 'Red').draw()
widgets.SmoothRectangle(canvas, 170,50, 240,70).draw()
kw = {'dx': 20, 'dy': 20, 'stipple': 'gray50', 'width': 3, 'outline': 'Blue'}
widgets.SmoothRectangle(canvas, 110,120, 190,190, **kw).draw()

root.mainloop()
```

A.6 Spring demonstration

This is a demonstration of how to use the `Spring` class found in the `widgets` module (chapter 10).

```
#!/bin/sh
""":"
exec python $0 ${1+"$@"}
"""

# Make sure the modules are found
import sys
sys.path[:0] = ['../']

import Tkinter
import Pmw
from modules import widgets
from math import *

class Gui:
    def __init__(self):
        root.bind('<q>', self.exit)
```


Demonstrating the options of the Spring class
angle:
45
0 45 90 135 180 225 270 315 360
noOfLoops:
5
1 2 3 4 5 6 7 8 9 10 11 12
displacement:
0.50
-1.00 -0.50 0.00 0.50 1.00
samples:
100
2 52 102 152 202 252
use standard ratio?
startEndLength:
12
0 10 20 30 40 50
use standard ratio?
width:
18
0 10 20 30 40 50
length:
209
50 75 100 125 150 175 200 225 250


Demonstrating the options of the Spring class
angle:
135
0 45 90 135 180 225 270 315 360
noOfLoops:
10
1 2 3 4 5 6 7 8 9 10 11 12
displacement:
0.40
-1.00 -0.50 0.00 0.50 1.00
samples:
175
2 52 102 152 202 252
use standard ratio?
startEndLength:
30
0 10 20 30 40 50
use standard ratio?
width:
30
0 10 20 30 40 50
length:
248
50 75 100 125 150 175 200 225 250

```
self.angle = 0
self.noOfLoops = 5
self.displacement = 0.0
self.samples = 100
self.startEndLength = 'use standard ratio'
self.width = 'use standard ratio'
self.length = 154
self.startEndLengthRatio = Tkinter.IntVar()
self.startEndLengthRatio.set('1')
self.widthRatio = Tkinter.IntVar()
self.widthRatio.set('1')


# Create widgets
args = (root, 'angle:', 0, 360, 45, 5, self.angle, self.change_angle)
scale1 = self.create_scale(*args)

args = (root, 'noOfLoops:', 1, 12, 1, 1, self.noOfLoops,
        self.change_noOfLoops)
scale2 = self.create_scale(*args)

args = (root, 'displacement:', -1.0, 1.0, 0.5, .01, self.displacement,
        self.change_displacement)
scale3 = self.create_scale(*args)

args = (root,'samples:', 2, 252, 50, 1, self.samples,
        self.change_samples)
scale4 = self.create_scale(*args)

frame1 = Tkinter.Frame(root,
        borderwidth = 2,
        relief = 'groove')

button1 = Tkinter.Checkbutton(frame1,
        command = self.change_startEndLength_state,
        text = 'use standard ratio?',
        variable = self.startEndLengthRatio)

args = (frame1, 'startEndLength:', 0, 50, 10, 1, 12,
        self.change_startEndLength)
self.startEndLengthScale = self.create_scale(*args)
self.startEndLengthScale.configure(state = 'disabled', fg = 'Gray60')

frame2 = Tkinter.Frame(root,
        borderwidth = 2,
        relief = 'groove')

button2 = Tkinter.Checkbutton(frame2,
        command = self.change_width_state,
        text = 'use standard ratio?',
        variable = self.widthRatio)

args = (frame2, 'width:', 0, 50, 10, 1, 18, self.change_width)
self.widthScale = self.create_scale(*args)
self.widthScale.configure(state = 'disabled', fg = 'Gray60')

args = (root, 'length:', 50, 250, 25, 1, self.length,
        self.change_length)
```

```
        scale5 = self.create_scale(*args)
        scale5.configure(length = 600)

        self.canvas = Tkinter.Canvas(root,
                borderwidth=2,
                relief='sunken',
                height = 500,
                width = 600)

        # Position widgets
        scale1.grid(row = 0, column = 0)
        scale2.grid(row = 0, column = 1)
        scale3.grid(row = 1, column = 0)
        scale4.grid(row = 1, column = 1)
        frame1.grid(row = 2, column = 0)
        button1.grid(row = 0, column = 0)
        self.startEndLengthScale.grid(row = 1, column = 0)
        frame2.grid(row = 2, column = 1)
        button2.grid(row = 0, column = 0)
        self.widthScale.grid(row = 1, column = 0)
        scale5.grid(row = 3, columnspan = 2)
        self.canvas.grid(row = 4, columnspan = 2, padx = 5, pady = 5)

        root.update()
        self.startEndLength = 'use standard ratio'
        self.width = 'use standard ratio'

    def create_scale(self, master, label, from_, to, tick, res, init, command):
        scale = Tkinter.Scale(master,
                command = command,
                from_ = from_,
                label = label,
                length = 285,
                orient = 'horizontal',
                resolution = res,
                to = to,
                tickinterval = tick,
                troughcolor = 'White')
        scale.set(init)
        return scale

    def create_and_draw_spring(self):
        self.spring = widgets.Spring(self.canvas,  300,250, self.length,
                angle = self.angle,
                noOfLoops = self.noOfLoops,
                samples = self.samples,
                startEndLength = self.startEndLength,
                width = self.width)
        self.spring.draw(self.displacement)

    def change_angle(self, angle):
        self.angle = 2*pi*float(angle)/360
        self.spring.angle = self.angle
        self.spring.draw(self.displacement)

    def change_noOfLoops(self, noOfLoops):
        self.noOfLoops = int(noOfLoops)
        self.create_and_draw_spring()
```

```
    def change_displacement(self, displacement):
        self.displacement = float(displacement)
        self.spring.draw(self.displacement)

    def change_samples(self, samples):
        self.samples = int(samples)
        self.spring.samples = self.samples
        self.spring.draw(self.displacement)

    def change_startEndLength(self, startEndLength):
        self.startEndLength = int(startEndLength)
        self.create_and_draw_spring()

    def change_startEndLength_state(self):
        if self.startEndLengthRatio.get():
            state = 'disabled'
            colour = 'Gray60'
            self.startEndLength = 'use standard ratio'
        else:
            state = 'normal'
            colour = 'Black'
            self.startEndLength = int(self.startEndLengthScale.get())
        self.startEndLengthScale.configure(state = state, fg = colour)
        self.create_and_draw_spring()

    def change_width(self, width):
        self.width = int(width)
        self.create_and_draw_spring()

    def change_width_state(self):
        if self.widthRatio.get():
            state = 'disabled'
            colour = 'Gray60'
            self.width = 'use standard ratio'
        else:
            state = 'normal'
            colour = 'Black'
            self.width = int(self.widthScale.get())
        self.widthScale.configure(state = state, fg = colour)
        self.create_and_draw_spring()

    def change_length(self, length):
        self.length = int(length)
        self.create_and_draw_spring()

    def exit(self, event = None):
        root.destroy()

#----
root = Tkinter.Tk()
root.title('Demonstrating the options of the Spring class')
Gui()
root.mainloop()
```

A.7 Two spring types

This is a test to find the best of two different spring types. The left spring (see the figures A.1 - A.3) is made with the following parametric formula

$$\begin{aligned} x(t) &= -a\cos t, \\ y(t) &= bt + c\sin t, \end{aligned}$$

and the right spring with

$$\begin{aligned} x(t) &= -a\sin t, \\ y(t) &= bt, \end{aligned}$$

for some suitable values of the contants a, b and c, and on some interval $[t_0, t_1]$. The first formula was preferred in the implementation (the left spring simply looks better).

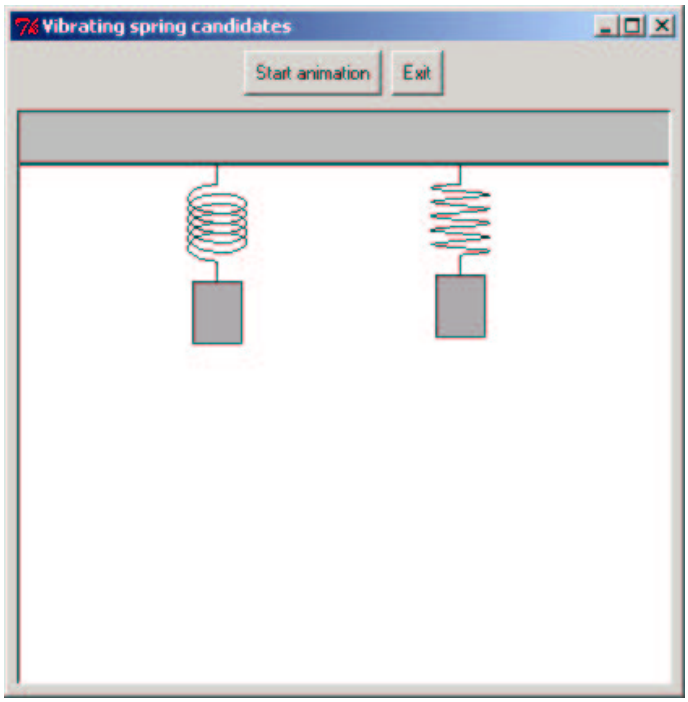


Figure A.1: Maximum compressed state.

```
#!/bin/sh
""":"
exec python $0 ${1+"$@"}
"""

import Tkinter
import Pmw
import re
from math import *
```

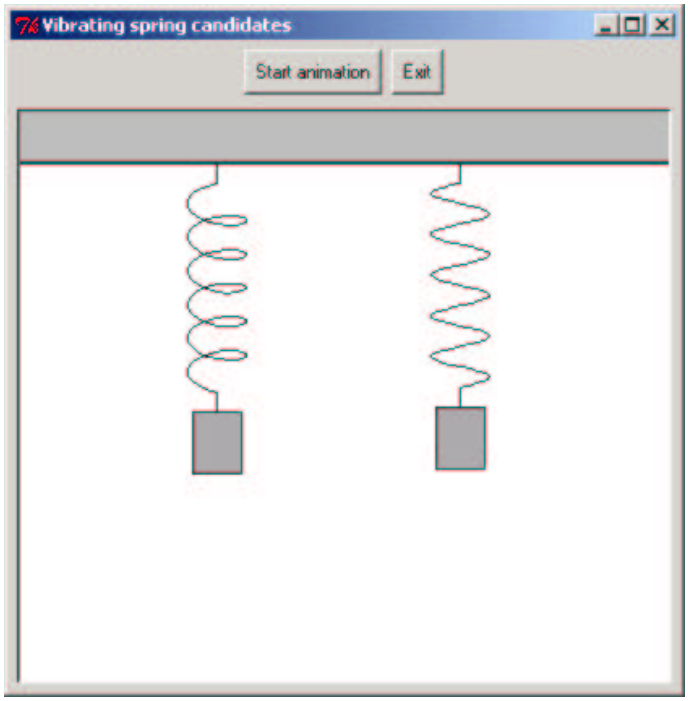


Figure A.2: The springs in equilibrium.

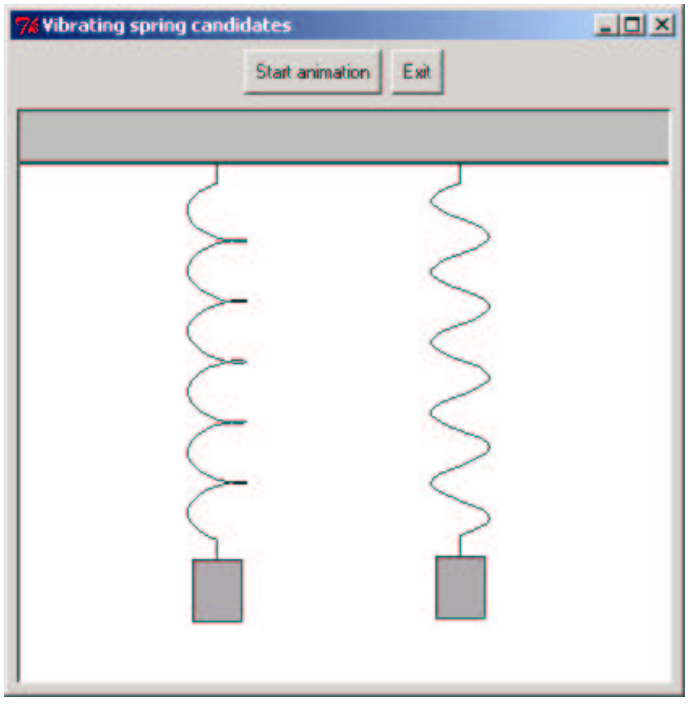


Figure A.3: Maximum outstretched state.

```
class Gui:
    def __init__(self):
        root.bind('<q>', self.exit)

        buttonbox = Pmw.ButtonBox(root)
        buttonbox.add('Start animation', command = self.start_animation)
        buttonbox.add('Exit', command = self.exit)
        buttonbox.pack(padx = 2, pady = 2)

        self.canvas = Tkinter.Canvas(root,
                background = 'White',
                borderwidth = 2,
                height = 350,
                relief = 'sunken',
                width = 400)
        self.canvas.pack(padx = 2, pady = 2)
        self.canvas.create_rectangle(-5,-5, 405,35, width = 2, fill = 'gray75')

        self.spring1 = Spring1()
        self.draw_spring(self.spring1)

        self.spring2 = Spring2()
        self.draw_spring(self.spring2, x0 = 275)

    def start_animation(self):
        t = 0
        while t <= 5*pi:
            t = t + .15
            y = eval(re.sub('t', str(t), 'sin(t)'))

            self.spring1.new_displacement(y)
            self.draw_spring(self.spring1)

            self.spring2.new_displacement(y)
            self.draw_spring(self.spring2, x0 = 275)

            root.update()

    def draw_spring(self, spring, x0 = 125):
        self.canvas.delete(spring.tag)
        values = spring.calculate_values()
        y0 = 35
        cy = 47
        last_y = values[0][1]
        coords = []
        for x,y in values:
            cx = x0 + x
            cy = cy + y - last_y
            last_y = y
            coords.extend((cx,cy))

        self.canvas.create_line(x0,y0, x0,y0+12, tags = spring.tag)
        self.canvas.create_line(coords, tags = spring.tag)
        self.canvas.create_line(x0,cy, x0,cy+12, tags = spring.tag)
        self.canvas.create_rectangle(x0-15,cy+12, x0+15,cy+50,
                fill = 'DarkGray', tags = spring.tag)

    def exit(self, event = None):
        root.destroy()
```

```
class Spring1:
    def __init__(self):
        self.a = 18
        self.b = 3.3  # min: 0.6  max: 6.0    equilibrium: 3.3
        self.c = 7.5
        self.t0 = -pi/2
        self.t1 = 10*pi + 5*pi/8
        self.samples = 100
        self.tag = 'spring1'

    def new_displacement(self, y):
        self.b = 3.3 + y*2.7

    def calculate_values(self):
        self.x = '-' + str(self.a) + '*cos(t)'
        self.y = str(self.b) + '*t+' + str(self.c) + '*sin(t)'
        dt = (self.t1 - self.t0)/float(self.samples)
        values = []
        for n in range(self.samples):
            t = self.t0 + n*dt
            x = eval(re.sub('t', str(t), self.x))
            y = eval(re.sub('t', str(t), self.y))
            values.append((x,y))
        return values

class Spring2:
    def __init__(self):
        self.a = 18
        self.b = 4.0  # min: 1.0  max: 7.0    equilibrium: 4.0
        self.t0 = 0
        self.t1 = 10*pi + pi/9
        self.samples = 100
        self.tag = 'spring2'

    def new_displacement(self, y):
        self.b = 4.0 + y*3.0

    def calculate_values(self):
        self.x = '-' + str(self.a) + '*sin(t)'
        self.y = str(self.b) + '*t'
        dt = (self.t1 - self.t0)/float(self.samples)
        values = []
        for n in range(self.samples):
            t = self.t0 + n*dt
            x = eval(re.sub('t', str(t), self.x))
            y = eval(re.sub('t', str(t), self.y))
            values.append((x,y))
        return values

#----
root = Tkinter.Tk()
root.title('Vibrating spring candidates')
Gui()
root.mainloop()
```

Bibliography

[1] Python official website. URL: `www.python.org`

[2] Python Library Reference. URL: `www.python.org/doc/current/lib/lib.html`

[3] F. Lundh. *An Introduction to Tkinter*. URL: `www.pyhonware.com/library/tkinter/introduction/index.htm`

[4] Pmw megawidgets. URL: `pmw.sourceforge.net/`

[5] D. M. Beazley. *Python essential reference*. New Riders Publishsing, 2000.

[6] J. E. Grayson. *Python and Tkinter programming*. Manning Publications, 2000.

[7] F. M. White. *Viscous fluid flow*. McGraw-Hill, 1991.

[8] H. P. Langtangen. *Forelesninger i ME211*. University of Oslo, 1997.

[9] G. F. Simmons. *Differential equations with applications and hostorical notes*. McGraw-Hill, 1991.

[10] M. T. Goodrich and R. Tamassia. *Data Structures and Algorithms in Java*. John Wiley & Sons, 1998.

[11] M. A. Weiss. *Data Strucutures and Algorithm Analysis*. The Benjamin/Cummings Publishing Company, 1995.

[12] D. Kincaid and W. Cheney. *Numerical Analysis*. Brooks/Cole Publishing, 1996.

[13] H. P. Langtangen *Scripting Techniques in Computational Science*. University of Oslo, 2000.

[14] J. K. Ousterhout. *Scripting: Higher-level programming for the 21st centuray*. IEEE Computer Magazine, 1998.