Python for Scientific Computations and Control

Project Report

Georg Malte Kauf

Contents

1 Introduction

Aim of this report is to show and explain the signal processing application *SigPro* which has been programmed in python using the Tkinter toolkit. It was made to provide basic signal processing in a user-friendly graphical user interface.

2 Tkinter Graphical User Interface

2.1 Creating the Master Frame

The Tkinter python module is an easily to use but yet powerful graphical user interface toolkit. To create a graphical user interface (GUI) it is necessary to create a master frame in which all the other widgets are placed. This is done by the following code lines:

```
frame = Frame(master)
frame.grid()
```

which are referring to a root widget. The whole application is packed into a class which is recommended. While creating an instance of this class by

```
root = Tk()
gui_frame = mainFrame(root)
```

the root widget is passed to the class as an argument and is within the class called *master* as it can be seen here:

```
def __init__(self, master):
```

which is the code line which starts the initial method while creating an instance of the class *mainFrame*. In the parent widget *Frame(master)* all children widgets are placed.

Finally to execute and show all Tkinter widgets created by the root widget *root* it is necessary to call a main loop.

```
root.mainloop()
```

2.2 Using Grid Geometry Manager

Tkinter provides simple but effective tools like the grid geometry manager which has been used in this application. With this manager all used widgets can easily be arranged using a column and row grid with the possibility to extend the widget over several columns or rows. For example the used checkbutton is arranged with the code line:

```
self.checkbut.grid(row = 9, column = 0, columnspan = 2)
```

where the options *row* and *column* define the position in the grid and *columnspan* extends the button over several columns which is also possible for rows. When the option *sticky* is used the widget's orientation in its grid cell can be defined. Every widget must be placed with the grid manager to be seen in the Tkinter main frame. There are other tools to place widgets within a master widget but only one type should be used since problems will occur using two different types of geometry manager.

Only the grid geometry manager was used in this application except for the prompt messages where *pack* was used. But that is possible since they were not placed in the same master frame where *grid* was used.

2.3 Implementing the Menu Bar

Essential for every GUI is some kind of menu where the user is guided by. That is why also this GUI has a menu bar with different tabs and functions. To implement these in the main frame the following procedure is used. A master widget for the menu is created

```
menubar = menu(Master)
```

where all sub menu functions are implemented in. Then there is a pulldown menu created and after that there are commands added, e.g.

```
editmenu = Menu(menubar, tearoff=0)
editmenu.add_command(label="Undo", command=self.undo)
editmenu.add_command(label="Redo", command=self.redo)
menubar.add_cascade(label="Edit", menu=editmenu)
```

for the edit pulldown menu. The same way more menu tabs can be added. In the end the menu must be added to the master frame.

```
master.config(menu=menubar)
```

2.4 Implementing *Label*, *Button*, *Checkbutton* and *Entry* Widgets

The *Label* widget is a very easy to implement one. An instance of the *Label* class is created and after that with the grid geometry manager added to the master frame. For example

```
label1 = Label(master, text="Loading and Saving Data")
label1.grid(row=3, column=0, columnspan=4)
```

whereas more optional arguments can be passed to change e.g. the script color.

The *Button* widget is used to execute a defined command when the user clicks on it. The command to create a button is for example

```
self.button = Button(
    master, text='Save Data',
    command=self.save, width=20
    )
```

where the argument *width* defines the button width which is also possible for the hight and the *command* argument defines the method or the like which should be executed. By the grid manager the button is placed within the frame.

Similar is the procedure to implement a checkbutton although the variable which can be changed by activating or deactivating the checkbutton must be defined before.

```
self.checkvar = IntVar()
```

and then the actual checkbutton is created.

```
self.checkbut = Checkbutton(
    master, text="Using Orignially Loaded Data",
    variable=self.checkvar
    )
```

An entry field is used to let the user insert for example a file name or a number. The variable can then later be accessed. An example is

```
self.e_resampling = Entry(master)
```

which is later accessed by the command

```
self.e_resampling.get()
```

2.5 Implementing a *Matplotlib* Figure and *Toolbar*

The *Matplotlib* figure in the program is one of the most important features. The user can immediately see how the current signal looks like. To implement the figure the module *matplotlib.backends.backend_tkagg* imported as *tkagg* is used which also creates the toolbar of the *matplotlib* figure. The *matplotlib.figure* module is imported as *mplfig*. A *canvas* is used to display the plot figure after defining a plot figure name

```
self.fig = mplfig.Figure()
self.axes = self.fig.add_subplot(111)
self.axes.grid(TRUE)
self.canvas = tkagg.FigureCanvasTkAgg(self.fig, master = master)
self.canvas.get_tk_widget().grid(row=0, column=4, rowspan=25,)
self.canvas.draw()
```

and then it is placed in the grid and drawn.

The navigation toolbar which is necessary to zoom in or out in the plot figure or even to save the figure is also created with *matplotlib.backends.backend_tkagg* in a previously created toolbar frame

```
self.toolbarframe=Frame(master)
self.toolbarframe.grid(row=25, column=4, sticky='NWE')
self.toolbar = tkagg.NavigationToolbar2TkAgg(self.canvas,
    self.toolbarframe)
self.toolbar.grid(row=26,column=4, sticky='NWE')
```

and positioned by the grid manager.

3 Program's Functions and Methods

3.1 Loading and Saving Data

Every command is executed by the user by clicking either on the button or in the menu button. Doing so the method is called. Before the signal processing can start data has to be loaded into the program. This tool was made for processing data with two columns or rows so for example one row of time data and one row of values. The data is loaded by

```
x = loadtxt(varname)
```

where *varname* is the content of a entry field. After that time and values are split and finally saved to current and originally loaded data. It is possible for the user to decide if he wants to use the data loaded in the beginning or to continuously process the signal by clicking on the checkbutton. Also the data from one step before is always saved so that the user can execute an *undo* command to revoke one command. It has to be said that the file has not to be a .txt file.

Furthermore it is possible to browse a file in the directories by clicking in the menu on *open* and it is possible to save data to a specific file by browsing by clicking in the menu on *save*. Otherwise by clicking in the window on *save data* the file will be saved in the current directory.

3.2 Signal Processing Tools

3.2.1 Resampling

Very important for signal processing and following usage of data is it to reduce data volume or simplify it. For this aim the resampling function was implemented which just uses every e.g. second sample as it is wished by the user who can type the resampling rate directly into the entry field.

3.2.2 Moving Average Filter

A moving average filter is a tool to reduce noise in a signal and making the signal smoother. This is done by calculating the mean value of the next n samples of every sample. The number n can be typed by the user in the corresponding entry field.

3.2.3 Coarse Graining Filter

A coarse graining filter is a combination of moving average filter and resampling. It is simultaneously done and reduces noise and making the signal very smooth as the name says.

3.2.4 Fast Fourier Transformation

The fast Fourier transformation is a powerful tool to analyse a signal and to compare it to others. The mainly used frequencies of the analysed signal are shown in the frequency spectrum when this function is used.

3.2.5 Autocorrelation Analysis

The correlation coefficient is a value between -1 and 1 and indicates how much two signals are linear depending on each other. A value of 1 means that the two signals are absolutely linear dependent and -1 means a absolute reverse dependency. The autocorrelation function is the correlation coefficient of one and the same signal for different lags meaning for different distances between samples. Thus the periodicity can easily be seen in the autocorrelation function plot which is displayed pressing the autocorrelation button.

3.3 Other Functions

Some further functions of the program are as mentioned before the loading and saving feature of processed signals even with the option to browse between directories. Also mentioned was the option to undo every processing step and use the previously loaded data by activating the checkbutton.

Very useful are the *undo* and *redo* functions. The user has with it the possibility to revoke the last step of processing or redo it again. This step is only possible once though since it should be sufficient only to reach the last state to undo one command and also the last data need to be saved to a variable.

Furthermore it is possible for the user to call this report as a manual by pressing the manual button in the menu and with that reading this documentation including the source code.

The plot figure and the corresponding toolbar are of course also fully functional.

4 Potential for Improvement

In this section some aspects shall be discussed that can be improved in the program. One thing that is of course the range of functions of the program. Right now there are only a few functions and it is only possible to process one signal at once. Later versions of this program should provide more complex processing functions.

Another aspect is the layout. At current state there was done almost no special effort in making the layout more than functional. Thus there is no change of background color or script font. This could be improved to make the program more user-friendly.

Maybe the most important and difficult to change aspect is the usage of global variables in the program. The loaded data, the current data, and the undo variables are all defined as global so every method can access them easily. But especially for memory usage aspects or even because of the chance to overwrite them it is not recommended to use global variables. Since the size of the data and with that the space in memory reserved for these variables changes all the time when loading or resampling the data global variables should not be used. The next version of this program must consider this problem.

Appendix

Python Code

```
## python gui application using Tkinter

## main

from Tkinter import *
from numpy import *
import matplotlib.pyplot as plt
import matplotlib.figure as mplfig
import matplotlib.backends.backend_tkagg as tkagg
import os
import tkFileDialog
from scipy import fft


# global variables, careful!
# loaded original data
x1 = 0  # time
x2 = 0  # values
# current used data
c1 = 0  # time
c2 = 0  # value
# undo variable
u1 = 0  # time
u2 = 0  # value
undo = False

class mainFrame:

    def __init__(self, master):

        frame = Frame(master)
        frame.grid()

        def hello():
            msg = "Welcome to SigPro! This Application is made for " \
                "easy signal processing and basic investigations of data!"
            print msg

        menubar = Menu(master)
```

```
# create a pulldown menu, and add it to the menu bar
filemenu = Menu(menubar, tearoff=0)
filemenu.add_command(label="Open", command=self.browsingopenfile)
filemenu.add_command(label="Save", command=self.browsingsavefile)
filemenu.add_separator()
filemenu.add_command(label="Exit", command=master.quit)
menubar.add_cascade(label="File", menu=filemenu)

# create more pulldown menus
editmenu = Menu(menubar, tearoff=0)
editmenu.add_command(label="Undo", command=self.undo)
editmenu.add_command(label="Redo", command=self.redo)
menubar.add_cascade(label="Edit", menu=editmenu)

helpmenu = Menu(menubar, tearoff=0)
helpmenu.add_command(label="About", command=hello)
helpmenu.add_command(label="Manual", command=self.openmanual)
menubar.add_cascade(label="Help", menu=helpmenu)

# display the menu
master.config(menu=menubar)

## inplementing labels and buttons
label1 = Label(master, text="Loading and Saving Data")
label1.grid(row=3, column=0, columnspan=4)

# load data widgets
label0 = Label(master, text="Filename: ")
label0.grid(row=4)
self.e_file = Entry(master)
self.e_file.grid(row=4, column=1)
self.button = Button(
    master, text='Load', command=self.loaddata, width=20
    )
self.button.grid(row=4, column=2)

# saving data widgets
label2 = Label(master, text="Filename: ")
label2.grid(row=5, column=0)
self.e_save = Entry(master)
self.e_save.grid(row=5, column=1)
self.button = Button(
    master, text='Save Data',
    command=self.save, width=20
```

```
    )
self.button.grid(row=5, column=2)

# checkbutton
self.checkvar = IntVar()
self.checkbut = Checkbutton(
    master, text="Using Origninally Loaded Data",
    variable=self.checkvar
    )
self.checkbut.grid(row = 9, column = 0, columnspan = 2)

# Filtering widgets
label3 = Label(master, text="Signal Filtering Tools")
label3.grid(row=15, columnspan=4)

# resampling
label3_1 = Label(master, text="Resampling")
label3_1.grid(row=16)
label3_1_1 = Label(master, text="Resampling Rate: ")
label3_1_1.grid(row=16, column=1)
self.e_resampling = Entry(master)
self.e_resampling.grid(row=16, column=2)
self.button = Button(
    master, text='Resample', command=self.resample, width=20
    )
self.button.grid(row=16, column=3)

# moving average filter
label3_2 = Label(master, text="Moving Average Filter")
label3_2.grid(row=17)
label3_2_1 = Label(master, text="n = ")
label3_2_1.grid(row=17, column=1)
self.e_maf = Entry(master)
self.e_maf.grid(row=17, column=2)
self.button = Button(
    master, text='Filter', command=self.mafilter, width=20
    )
self.button.grid(row=17, column=3)

# coarse graining filter
label3_3 = Label(master, text="Coarse Graining Filter")
label3_3.grid(row=18)
label3_3_1 = Label(master, text="tau = ")
label3_3_1.grid(row=18, column=1)
```

```
        self.e_cgrain = Entry(master)
        self.e_cgrain.grid(row=18, column=2)
        self.button = Button(
            master, text='Filter', command=self.fcgrain, width=20
            )
        self.button.grid(row=18, column=3)

        # FFT widgets
        label4 = Label(master, text="Fast Fourier Transformation")
        label4.grid(row=22)
        self.button = Button(
            master, text='FFT', command=self.fastfourier, width=20
            )
        self.button.grid(row=22, column=2)

        # autocorrelation widgets
        label5 = Label(master, text="Autocorrelation Signal")
        label5.grid(row=24)
        self.button = Button(
            master, text='Autocorrelation', command=self.autocorr, width=20
            )
        self.button.grid(row=24, column=2)

        # creating matplotlib figure
        self.fig = mplfig.Figure()
        self.axes = self.fig.add_subplot(111)
        self.axes.grid(TRUE)
        self.canvas = tkagg.FigureCanvasTkAgg(self.fig, master = master)
        self.canvas.get_tk_widget().grid(row=0, column=4, rowspan=25,)
        self.canvas.draw()

        # creating toolbar frame
        self.toolbarframe=Frame(master)
        self.toolbarframe.grid(row=25, column=4, sticky='NWE')
        # creating a toolbar for saving, zooming etc. (matplotlib standard)
        self.toolbar = tkagg.NavigationToolbar2TkAgg(self.canvas,
            self.toolbarframe)
        self.toolbar.grid(row=26,column=4, sticky='NWE')

    # methods
    # loading data
    def loaddata(self):
        global x1, x2, c1, c2
        varname = self.e_file.get()
```

```
        self.saveundovar()
        #print varname
        x = loadtxt(varname)
        x1 = x[:,0]
        x2 = x[:,1]
        self.plotit(x1, x2)
        # saving variables to global current variables
        c1 = x1
        c2 = x2

    def plotit(self, x, y):
        self.axes.cla()  # clear all, delete former plot
        self.axes.plot(x, y)
        self.axes.grid(TRUE)
        self.canvas.draw()  # draw figure

    def save(self):
        name = self.e_save.get()  # getting file name
        d = open(name, 'w')  # open file
        savetxt(d, array([c1, c2]).T)  # save current data
        d.close()  # closing file

    def resample(self):
        global c1, c2
        [t,y] = self.choosedata()
        resrate = int(str(self.e_resampling.get()))  # getting sample rate
        if (resrate <= 0):
            self.promptmessage('math')
        else:
            self.saveundovar()
            r1 = zeros(len(range(0, len(t), resrate)))
            r2 = zeros(len(range(0, len(y), resrate)))
            for i in range(0, len(t), resrate):  # resampling
                j = i / resrate
                r1[j] = t[i]
                r2[j] = y[i]
            self.plotit(r1, r2)
            # saving variables to global current variables
            c1 = r1
            c2 = r2

    def mafilter(self):
        global c1, c2
        [t,y] = self.choosedata()
```

```
        n = int(str(self.e_maf.get()))  # number of samples to use for mean
        if (n <= 0):
            self.promptmessage('math')
        else:
            self.saveundovar()
            z = zeros(len(y))
            for i in range(0, len(y)):
                z[i] = mean(y[i:i + n - 1])
            self.plotit(t, z)
            # saving variables to global current variables
            c1 = t
            c2 = z

    def fcgrain(self):
        global c1, c2
        [t,y] = self.choosedata()
        tau = int(str(self.e_cgrain.get()))  # getting scaling factor
        if (tau <= 0):
            self.promptmessage('math')
        else:
            self.saveundovar()
            N = len(t)  # data lenght
            Ntau = N / tau  # grained data length
            xtau = zeros(Ntau)
            for j in range(0, Ntau):  # coarse graining
                xtau[j] = mean(y[j * tau:(j + 1) * tau - 1])
            ttau = zeros(len(xtau))
            for i in range(0, len(xtau)):
                ttau[i] = tau * t[i]  # time reconstruction
            self.plotit(ttau, xtau)
            # saving variables to global current variables
            c1 = ttau
            c2 = xtau

    def fastfourier(self):
        global c1, c2
        self.saveundovar()
        [t,y] = self.choosedata()
        frespec = abs(fft(y))
        self.axes.cla()  # clear all, delete former plot
        self.axes.plot(frespec)
        # show half of the spectrum
        self.axes.set_xlim(([0, len(frespec) / 2]))
        # setting limit y-axis
```

```
        self.axes.set_ylim(([0, max(frespec[2:])]))
        self.axes.grid(TRUE)
        self.canvas.draw()  # draw figure
        # saving variables to global current variables
        c1 = range(int(ceil(0.5 * len(t))))
        c2 = frespec[0:int(ceil(0.5 * len(t)))]

    def autocorr(self):
        global c1, c2
        self.saveundovar()
        [t,y] = self.choosedata()
        r = zeros(int(ceil(0.5 * len(t))))
        r[0] = 1
        for lag in range(1, len(r)):
            x = y[lag - 1:-1]
            z = y[0:-lag]
            meanx = mean(x)  # mean value for x
            meanz = mean(z)  # mean value for z
            stdx = std(x)  # standard deviation for x
            stdz = std(z)  # standard deviation for z
            vp = zeros(len(x))
            for i in range(0, len(x)):
                vp[i] = (x[i] - meanx) * (z[i] - meanz)
            r[lag] = sum(vp) / len(vp) / stdx / stdz
        c1 = range(0, len(r))
        c2 = r
        self.plotit(c1, c2)

    def openmanual(self):
        os.popen('.\BoardingPass.pdf')

    def browsingopenfile(self):
        global c1, c2, x1, x2
        self.saveundovar()
        fnameopen = tkFileDialog.askopenfilename(
            filetypes = [("txt-files", ".txt"),("all files",".*")]
            )  # browse file to be opened
        x = loadtxt(fnameopen)
        # initially loaded data
        x1 = x[:,0]
        x2 = x[:,1]
        self.plotit(x1, x2)
        # current data
        c1 = x1
```

```
        c2 = x2

    def browsingsavefile(self):
        global c1, c2
        fnamesave = tkFileDialog.asksaveasfilename(
            filetypes = [("all", ".*")]
            )
        d = open(fname, 'w')  # open file
        savetxt(d, array([c1, c2]).T)  # save data
        d.close()  # closing file

    def choosedata(self):
        global c1, c2
        if (self.checkvar.get() == 1):
            t = x1
            y = x2
        else:
            t = c1
            y = c2
        return [t,y]

    def saveundovar(self):
        global c1, c2, u1, u2, undo
        u1 = c1
        u2 = c2
        undo = False

    def undo(self):
        global c1, c2, u1, u2, undo
        if (undo == False):
            auxvar1 = c1
            auxvar2 = c2
            c1 = u1
            c2 = u2
            u1 = auxvar1
            u2 = auxvar2
            undo = True
            self.plotit(c1, c2)
        else:
            self.promptmessage('undo')

    def redo(self):
        global c1, c2, u1, u2, undo
        if (undo == True):
```

```
            auxvar1 = c1
            auxvar2 = c2
            c1 = u1
            c2 = u2
            u1 = auxvar1
            u2 = auxvar2
            undo = False
            self.plotit(c1, c2)
        else:
            self.promptmessage('undo')

    def promptmessage(self, reason):
        top = Toplevel(
            )
        top.title('Error!')
        if (reason == 'math'):
            msgtext = 'Use a non-zero, non-negative integer value!'
        elif (reason == 'undo'):
            msgtext = 'undo/redo function has already been used!'
        else:
            msgtext = 'Unexpected Error!'
        msg = Message(top, text=msgtext, fg = 'red')
        msg.pack()
        promptbutton = Button(top, text='OK', command=top.destroy)
        promptbutton.pack()

root = Tk()

gui_frame = mainFrame(root)

root.mainloop()
root.destroy()
```