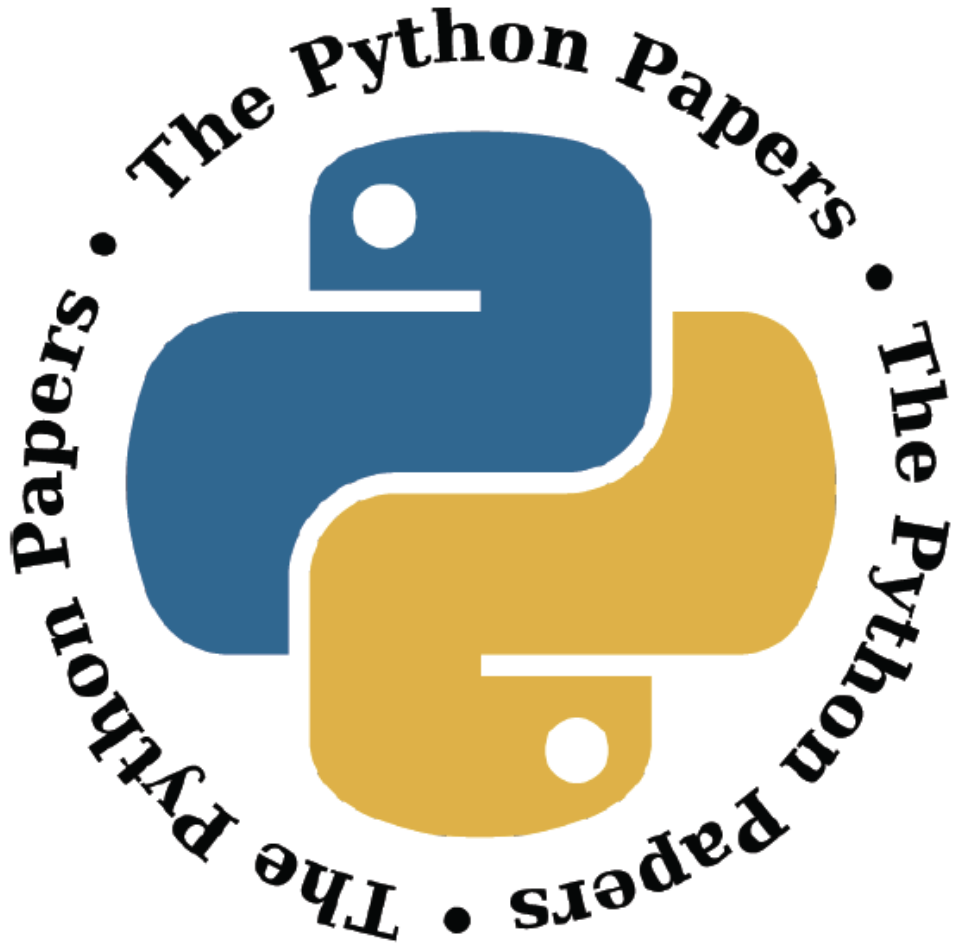
The Python Papers



Volume 3, Issue 2

pythonpapers.org

Journal Information

The Python Papers

ISSN: 1834-3147

Editors

Referencing Information

Articles from this edition of this journal may be referenced as follows:

Author, "Title" (2007) The Python Papers, Volume N, Issue M, pp. m:n
e.g. Maurice Ling, "Firebird Database Backup by Serialized Database Table Dump" (2007) The Python Papers, Volume 2, Issue 1, pp. 7:15.

Copyright Information

The Python Papers was first published in 2006 in Melbourne, Australia.

Referees

An academic peer-review was performed on all academic articles.
A list of reviewers will be published in each December issue.
This has been done in order to ensure the anonymity of reviewers for each paper.

Editorial

Maurice Ling and Tennessee Leeuwenburg

Tennessee> Readers may have noticed that we did not follow our usual publishing cycle, releasing this issue later in the year than usual. This reflects time taken to migrate *The Python Papers* to a new journal management system, as well as significant changes happening on the editorial committee. Maurice Ling is now Editor-In-Chief, but I am continuing my involvement by sharing this role as Co-Editor-In-Chief. This is due to the significant workload involved in the Editor-In-Chief role, requiring many hours of work to assemble each issue.

We also welcome Sarah Mount and Guy Kloss to the editorial committee; Sarah will be working to support our academic sections. With the leadership of Maurice, and the assistance of Sarah and Guy, I will hopefully be able to bring additional attention to the industry and community aspects of the journal.

In a blatantly Australian plug, I would also like to draw reader's attention to the Open Source Developer's Conference 2008 (http://osdc.com.au/2008/index.html). This is an excellent local conference for software developers, with a lot of Python content.

Maurice> Hi there, dear readers. Sorry for making everyone wait so long for this issue. As Tennessee had mentioned, there are a number of changes in *The Python Papers* which I will explain here.

Firstly, in an attempt to widen our reach, we are now using Open Journal System (http://pkp.sfu.ca/?q=ojs) to manage *The Python Papers*. As a result, *TPP* is now indexed in:

- OAIster (http://www.oaister.org/browse/browsep.html)
- Public Knowledge Project (http://pkp.sfu.ca/harvester2/demo/index.php/browse)
- Google Scholar

  http://scholar.google.com/scholar?hl=en&lr=&q=&as_publication= Python+Papers&btnG=Search
- At the same time, we had ported all past issues of The Python Papers into this new system (http://ojs.pythonpapers.org/index.php/tpp).

Secondly, we had soft-launched 2 more periodicals – *The Python Papers Monographs* and *The Python Papers Source Codes*. *The Python Papers Monographs* (TPPM) will cater for monograph-length submissions which may include dissertations, conference proceedings, case studies and advanced-level lectures. *The Python Papers Source Codes* (TPPSC) is modeled after ACM Collected Algorithms and provides a collection of software and source codes, usually associated with papers published in *The Python Papers* and *The Python Papers Monograph*. Collectively, we had named these 3 periodicals as *The Python Papers Anthology* (TPPA, http://ojs.pythonpapers.org/) under the same editorial committee. At the same time, to make it easier for all readers, we will regularly synchronize users across these periodicals.

Thirdly, *TPP* will only be published 3 times a year instead of 4. This is in-line with our original ISSN registration. Hence, the 3 issues will be scheduled for April, August and December each year.

Fourthly, we are pleased to announce that we had signed up Ian Ozsvald as *TPP*'s regular columnist on "ShowMeDo Updates" where Ian will do a periodic updates on the new additions to ShowMeDo.

Lastly and perhaps most importantly, we welcome Sarah Mount and Guy Kloss into the editorial

committee. Sarah is a Senior Lecturer in School of Computing and Information Technology at the University of Wolverhampton, UK, and is also the co-author of "*Python for Rookies: A First Course in Programming*" published by Thomson Learning. Guy Kloss is a PhD candidate of computer science in Massey University in New Zealand with interest in grid computing and artificial intelligence.

Errata

The previous issue had a number of editorial mistakes, for which we convey our apologies to both readers and authors. The errors in Volume3, Issue 3 were:

- Correction to"showmedo-authors" by Ian Ozsvald on page 14:
  Horst Jens (http://showmedo.com/videos/?author=71) was previously featured in Volume 2, Issue 2 of The Python Papers. He has authored 25 videos with various students at spielendprogrammieren (www.spielendprogrammieren.at) and other schools on Python and open-source topics.
- Ian Ozsvald's name was misspelled in several places.
- Some difficulties in page formatting occurred as the journal grew too large for effective editing in our journal layout program. Specifically, some readers reported pages being overlaid on one another rather than having a proper page order. From this issue, individual articles may be downloaded which will at least provide alternative access should any further issues occur.

ShowMeDo Updates

Ian Ozsvald

ShowMeDo.com exists to teach you new skills using screencasts. We and over 100 open-source contributors make videos for the site, in total we have 730 screencasts for topics like Python, Inkscape, Open-Office, Perl and web-programming.

By visiting ShowMeDo you'll get to see how other people use a language like Python - which libraries they use, their development environment and the projects they favour. You'll find over 340 Python videos here: http://showmedo.com/videos/python

If you're a beginner to Python then you'll want access to strong, complete tutorials that teach you the necessary core skills. Our Club hosts a subset of the videos and is for paying subscribers, each week we add two new videos (we're at 102 and counting). These series include walk-throughs, examples and practical exercises. You can find out more about the Club here: http://showmedo.com/club

> "As Ian says somewhere, he's found, organized, and presented information that would take weeks or months to learn in the ordinary way. Even if your time is worth only $10 per hour, I figure the value in time saved is at least $800. The skills are priceless."
> - Vincent DiCarlo

In an earlier Python Papers (Volume Three, Issue One : March 2008) we took a tour of the tutorials by our main Python authors. In this issue we'll look at the Python videos that have been added since last time.

Free Python Videos

First, let's take a look at the free videos. Most of these have been contributed by our open-source authors - you could contribute too (learn more at the end of this article). By contributing you'll help to teach others about the power of Python.

Learning and Teaching Python
Gasto, Horst and Kirby have created 9 videos aimed at new Pythonistas. Gasto shows you how to get started with the language and the new 'set' datatype:

- Python from zero (2 videos), Gasto (http://showmedo.com/videos/series?name=cqZALPzt8)

Horst shows you how to use xturtle to draw turtle graphics:

- Python: xturtle, random and genetischer Algorithmus (German), Horst Jensâ, (http://showmedo.com/videos/series?name=yRIJviWLO)

- Kirby talks about using Python for math-education (which includes very nice graphics): Python for Math Teachers, (6 videos), Kirby Urner (http://showmedo.com/videos/series?name=JkD78HdCD)

> "Very helpful. I like the repetition you give using your examples it helps me more clearly understand what you are trying to say."

Simple installation videos have been created for users who are unsure of 'easy_install' and 'nosetests' - these tools should be core skills for all Pythonistas:

- Installing easy_install, Ian Ozsvald (http://showmedo.com/videos/video?name=2070000&fromSeriesID=207)
- Installing nosetests, Ian Ozsvald (http://showmedo.com/videos/video?name=2100000&fromSeriesID=210)

Florian then builds upon these skills with an introduction to unit-testing, including a look at unit-testing classes:

- Basic Unittesting with Python, Florian Mayer (http://showmedo.com/videos/video?name=2840000&fromSeriesID=284)

Percious further extends these skills - first he looks at 'virtualenv' and 'pastescript', these are excellent tools to help with building virtual development environments and deployable applications.

Next he covers unit-testing with 'nosetests' and execution coverage, finally he looks at how we can create robust documentation with 'sphinx'. This series of videos is called "Agile Development Tools in Python": http://showmedo.com/videos/series?name=mcfckfJ4w

> "Very cool. I learned a lot from that. Fluid demonstration, easy to follow. I really hadn't heard much about virtualenv before but it looks hugely useful. Will become part of my workflow methinks. Great screencast"
> -- Kyran

Python's use for web-development continues to grow with frameworks like Django, TurboGears and Pylons. We have some great videos on:

- Setting Up a Django Development Environment,(2 videos), ericflo (http://showmedo.com/videos/series?name=LY7fNbpc1)
- TurboGears 2, percious (http://showmedo.com/videos/video?name=2870000&fromSeriesID=287)
- Agile Python+Pylons to build pySvnManager, Jiang Xin (http://showmedo.com/videos/series?name=Y7x8NhL6h)

Related tools include 'twill' which allows us to act as if Python was a web-browser (great for testing web-sites with unit-testing!):

- Installing twill (http://showmedo.com/videos/video?name=2080000&fromSeriesID=208)

and a simple guide to using the simple XML-RPC remote procedure-calls library:

- Calling functions on a server using XMLRPC, Florian Mayer (http://showmedo.com/videos/video?name=2830000&fromSeriesID=283)

Whilst most of the videos in ShowMeDo are in English, some are in German. Lucas and Marius are extending a new introductory Python series aimed at German speakers:

- Programmierung mit Python, Lucas Holland and Marius Meinertâ, (4 videos) (http://showmedo.com/videos/series?name=47MFnkxOe)

Finally, Erik Thompson keeps adding to his excellent 'Developing emol!' series. Over 24 videos show you the complete development process from initial plans, through class development, to 3D GUI programming and bug-fixing:

- Developing emol!, Erik Thompson (http://showmedo.com/videos/series?name=vXJsRwlBX)

> "I can see that this was a labor of love. I simply love the way you present this video. You do not pretend to be a GL expert and go through details to make it simple to understand for a beginner. You are not trying to show how good you are but to make the subject easy to understand."

Club ShowMeDo (for Subscribers)

In the Club (http://showmedo.com/club) we're focusing on new and intermediate Python programmers. We're aiming to layout all the core skills, along with worked-examples of the major libraries, so a new programmer can quickly become productive. Our paying subscribers give us the resources to spend more time building these long, well-researched tutorials. We also give full Club access to all of our authors (you too can have full free access if you author a video for us).

For new Pythonistas our 'Python Beginners' sets introduce the core language, with some comparisons to other languags like Java and C++, using simple real-world examples:

- Python Beginners - What Does Python Look Like?, (6 videos), Ian Ozsvald (http://showmedo.com/videos/series?name=kmNu2xcp3)
- Python Beginners - Loops and Iteration, (4 videos), Ian Ozsvald (http://showmedo.com/videos/series?name=tIZs1K8h4)
- Python Beginners - Common Variables and Containers, (14 videos), Ian Ozsvald (http://showmedo.com/videos/series?name=AzsZ2afN2)

> "Excellent tutorial - Well thought out example, clear, with a logical progression that builds on itself. In the end the goal of the tut is attained very well."
> -- Hamish

Given Python's popularity for web-app development, we have a back-to-basics series on using the plain-old Common Gateway Interface:

- Introduction to Python web-programming: CGI, John Montgomery (http://showmedo.com/videos/series?name=gLK7tmgf4)

and this builds the viewer's basic knowledge so that tools like the Google App Engine make more sense:

- A Gentle Introduction to the Google App Engine Python SDK, (5 videos), Kyran Dale (http://showmedo.com/videos/series?name=FtBpzKiWL)

> "Glad to see you covering something quite current, and that does not require extensive programming experience."
> -- Mike Adams

Database development is always popular, Kyran introduces the topic with plenty of examples:

- An Introduction to Database Programming with Python, (5 videos), Kyran Dale (http://showmedo.com/videos/series?name=iNiVCfz5B)

> "Thank you a million again for such a great series."
> -- hessianmatrix

Finally, since Win32 COM programming such a tricky - but useful - subject, we have a series showing how to interact with MS Excel in Windows XP using just a few lines of Python:

- COM and Python with pyWin32, (6 videos), Ian Ozsvald (http://showmedo.com/videos/ series?name=KTN7wMXVN)

Would you like your tutorials to be seen by thousands of Python viewers?

We always welcome new authors - any topics that will educate our usual audience are wanted. You'll find instructions for recording your first screencast here:
http://showmedo.com/addVideoInstructions

and posting it to ShowMeDo is just as simple as adding it to YouTube (although we host videos at a larger size and much higher quality than YouTube!).

Our most popular topics include Django, wxPython and development environments but all topics have viewers in the long-tail and anything you record will find an audience:
http://showmedo.com/mostPopular

We have several forums and we're happy to help you create your first screencast, please do come and join us making more tutorials for Python.

Embedding a Python interpreter into your program

Alan J. Salmoni

Do you want to write your own Python interpreter that you can embed into your program rather than use IDLE? What, exactly, is the point of reinventing the wheel? Well, first of all, it's not reinventing it: it's re-designing it. This is a worthy task unless we prefer to use stone wheels.

While extending my statistics package, SalStat, I needed to embed an interpreter within the program. Primarily, this was for debugging purposes: with an interpreter, I could check on the state of variables and classes, and try to work out what was going on. But I found it such a useful tool to use as the main interface to the program that I extended it to include all sorts of weird functionality that may be fun. For example, instead of clicking through various screens to perform a range of descriptive statistics, I could script the entire thing.

```
for i in DO.varlist:
    descriptives([i], "sum","mean", "stdev")
```

This article describes how you can write your very own Python interpreter, and how it can be modified.

How does an interpreter work? Put simply, you create some kind of text interface (this could be a console, a simple multi-line edit control within a Tkinter window, or a wxPython frame with Scintilla – which is what ours is), and run the program. With a bit of magic, instead of clicking on buttons etc, you can examine variables in situ, modify things, run some code, and do pretty much what ever you would like as long as it's valid Python. The best thing is that you don't have to resort to exec or eval statements and there is persistence of variables. You can also import whatever variables you want. This was important because the interpreter needs to access those variables of interest to the user (whether the user is the programmer who wishes to debug the program, or the end-user who wishes to use the interpreter to issue commands).

I should point out that running a program from IDLE can probably achieve the same things; but it's fun to run your own, and sometimes IDLE is a bit heavy and this code is probably easier to customise and integrate. In addition, you can embed a very capable interpreter using iPython. The difference with the way described here is that when you make changes to the variables in the interpreter, they propagate back to the rest of the program.

The Code Module

An interpreter is based around the built-in 'code' module which isn't very large (it has two classes) but is extremely useful. This module "provides facilities to implement read-eval-print loops in Python".

The simplest way to implement an interpreter is using the InteractiveConsole class. This works much like Python when invoked from the command line. It is by far the simplest way to do things, but I don't use it because it stops all other operations until the interpreter is closed down. There are ways around this, but I consider the use of the other class, the InteractiveInterpreter class to be a better solution.

But if you want the InteractiveConsole, it's quite easy. Try this from IDLE or a script:

```
import code
terp = code.InteractiveConsole() # 'terp' is short for interpreter
terp.interact()
```

Wow. How clever. Here, I have managed to make a Python interpreter from a Python interpreter. Of course, I could have gone through all that drawn-out convoluted nonsense of typing just "python" at the command prompt instead (or even done nothing), so maybe not so clever. But remember that this can be called from within a program. One problem is that execution of the rest of the program halts until I kill the console. Still, it's a single import statement and two other statements to make it work. One fun thing to do might be to add arguments to the interact statement such as:

```
terp.interact("A kind of Python interpreter")
```

which brings up a different banner to the normal one. If you have an application and you want to advertise your organisation, you can insert what you want here. To either the InteractiveConsole or interact statement, you can also add local variables which will be discussed later.

We can get around this drawback by using the InteractiveInterpreter class which requires a little more work but is more flexible. First, we import the code module and then we have to put the InteractiveInterpreter class into the window. In this case, it's a wx styled text control (STC, also known as Scintilla which is great fun and extremely capable). I won't detail the STC here other than going through what is essential to setting up an interpreter.

First of all, we need to import the relevant modules: code for the interpreter, wx and wx.stc for the GUI widgets, sys to take stdout and stderr, and __main__ to deal with what objects are within the scope of the interpreter.

```
import wx # needed for the GUI
import wx.stc as stc # needed for the styled text control (Scintilla
component)
import code # where the interpreter is
import sys # needed for admin stuff
import __main__ # needed to import the variables you want to interact
with
```

Next, get the InteractiveInterpreter up and running. I did this by deriving a new class (called II) from this class.

```
class II(code.InteractiveInterpreter):
      def __init__(self, locals):
            code.InteractiveInterpreter.__init__(self, locals)

      def Runit(self, cmd):
            code.InteractiveInterpreter.runsource(self, cmd)
```

This class adds a new method called Runit which is not entirely necessary as runsource can be accessed directly. However, it is useful sometimes to derive classes so as to have custom functionality. What this class does is set up an interpreter and it can receive commands. The command output will be sent to stdout or stderr and these can be redirected to an appropriate place using the sys module (this is described later). When instantiated, this class requires an argument, locals. These are objects that should be visible to the interpreter and it is through this that you can define what should lie within the interpreter's scope.

So far, so good. You can type the above code into IDLE and instantiate the class. This will allow you to run simple code like this:

```
>>> x = II(None)
>>> x.Runit('print "hi"')
hi
```

which shows that the interpreter seems to work. These last two pieces of code are the core of the interpreter. However, for something more useful, you need to have some way of getting data in and out of this interpreter that is independent of IDLE, or in other words have a GUI with a customised Python interpreter running. For this paper, we are going to use a wxPython frame to hold the interpreter so we define a wx frame with a Scintilla component embedded:

```
class ecpintframe(wx.Frame):
      def __init__(self, *args, **kwds):
            kwds["size"] = (700,600)
            wx.Frame.__init__(self, *args, **kwds)
            self.ed = PySTC(self, -1)
```

This is the class definition of the wx frame that holds the STC. This is basic wxPython stuff. Then we have some code to instantiate the wxFrame which will cause it to appear once we have defined the PyTSC class:

```
if __name__ == '__main__':
      Ecpint = wx.PySimpleApp(0)
      win = ecpintframe(None, -1, "EcPint - Interactive intepreter")
      win.Show()
      Ecpint.MainLoop()
```

Then we define the Scintilla styled text control to do the hard work for us.

```
class PySTC(stc.StyledTextCtrl):
```

```
      def __init__(self, parent, ID, pos=(10,10), size=(700, 600),
style=0):
            stc.StyledTextCtrl.__init__(self, parent, ID, pos, size,
style)
            sys.stdout = self
            sys.stderr = self
            self.Bind(wx.EVT_KEY_DOWN, self.OnKeyPressed)
            KEY_RETURN = 13

      def SetInter(self, interpreter):
            self.inter = interpreter

      def write(self, ln):
            self.AppendTextUTF8('%s'%str(ln))
            self.GotoLine(self.GetLineCount())

    def OnKeyPressed(self, event):
        self.changed = True # records what's been typed in
        char = event.GetKeyCode() # get code of keypress
        if char == 13:
            lnno = self.GetCurrentLine()
            ln = self.GetLine(lnno)
            self.cmd = self.cmd + ln + '\r\n'
            self.NewLine()
            self.cmd = self.cmd.replace('\r\n','\n')
            self.inter.Runit(self.cmd)
            self.cmd = ''
            self.lastpos = self.GetCurrentPos()
        event.Skip() # ensure keypress is shown
```

This subclasses from the StyleTextCtrl class and changes a few attributes like making it take stdout and stderr (in other words, console output will be routed to this control). The SetInter method sets something (mysteriously called 'interpreter') as an attribute of the class. This is a direct link to the interpreter so that text typed into the STC can be sent straight there to be run. The write method outputs the text to the STC and is necessary if you want stdout and/or stderr output to be written there (they always look for a write method). The lines sys.stdout = self and sys.stderr = self redirect stdout and stderr respectively to the STC. This is a neat trick that means that all interpreter output is sent to the STC. You can of course decide to have errors printed elsewhere if that meets your needs better. The OnKeyPressed method catches keypresses and checks to see if they are the return key. If so, then the interpreter assumes that the user wishes to run the command; and the self.inter.Runit(self.cmd) does just that by sending the line just typed in to the interpreter. As we have already shown, stdout goes to the STC so you can use this already for input and ouput.

If you run this code, you will find that it works! If you type 'print "hi"' into the editor, it should print "hi" just underneath – we have a working interpreter!

However, there are problems with code blocks longer than one line. It is possible to get the interpreter to handle blocks at a time but this needs careful preparation of the data which means altering how the STC deals with typed-in data. Change the OnKeyPressed method above for this one below:

```
      def OnKeyPressed(self, event):
```

```
            self.changed = True # records what's been typed in
            char = event.GetKeyCode() # get code of keypress
            if (self.GetCurrentPos() < self.lastpos) and (char <314)
or (char > 317):
                  pass
                  # need to check for arrow keys in this
            elif char == 13:
                  lnno = self.GetCurrentLine()
                  ln = self.GetLine(lnno)
                  self.cmd = self.cmd + ln + '\r\n'
                  self.NewLine()
                  self.tabs = ln.count('\t')
                  if (ln.strip() == '') or ((self.tabs < 1) and (':'
not in ln)):
                        # record command in command list
                        self.cmd = self.cmd.replace('\r\n','\n')
                        # run command now
                        self.inter.Runit(self.cmd)
                        self.cmd = ''
                        self.lastpos = self.GetCurrentPos()
                  else:
                        if ':' in ln:
                              self.tabs = self.tabs + 1
                        self.AppendText('\t' * self.tabs)
                        # change cursor position now
                        p = self.GetLineIndentPosition(lnno + 1)
                        self.GotoPos(p)
            else:
                  event.Skip() # ensure keypress is shown
```

What this does is check whether the code would be expecting another line (say you typed in 'for x in range(5):'), and if so, it auto-indents using an extra tab for you. Purists will prefer spaces and this code can easily be changed to that if you wish. However, the above code will also indent if a colon is on the just-typed-in line so if there is a colon within a print statement, or a dictionary is set, then the code will be indented. This is far from a disaster and solved with backspacing; and you are encouraged to come up with a better method of knowing when to indent the next line.

Adding Local Variables

We can also add our own local variables. These are particularly useful for an embedded interpreter because they allow the interpreter to access your program's variables and objects. These aren't automatically included any of your program variables, functions, classes or methods and need to be explicitly specified. Adding them is easy. The interpreter has its own space which it calls __main__ (the same as the module that we imported at the start). This module contains references to a base set of objects. If you open the interpreter and import __main__, you can see what is available by typing dir(__main__). You should see objects concerning the interpreter ('Ecpint', 'I', 'II', 'PySTC', 'code' and others) as well as some others like __builtins__ or __doc__. The one that doesn't show up is the one of interest and this is __main__.__dict__ which is a dictionary that keeps the name and reference of objects. When you use an object, it looks in this dictionary to find out the object's reference, and can then call it. All we have to do is provide a reference to the objects we want to make available and provide a suitable name for them.

Let's assume you have an object called a dataObject which is instantiated using the name DO. We want that object to be visible to the interpreter (i.e., within the interpreter's scope). This is done with a simple one-line instruction:

```
__main__.__dict__["DO"] = DO
```

Of course, when this is issued, the DO needs to be visible. It could be passed (along with any other objects you want to be made visible) as an argument to the Inter class. Using this, you can make all of a program's objects available to the interpreter.

Interrupting Exceptions

The InteractiveInterpreter class also has a very useful method called 'showtraceback'. This is called when an exception is raised and normally shows the traceback that occurred along with the exception. It is possible to interrupt this and have some fun.

```
def showtraceback(self):
      type, value, tb = sys.exc_info()
      if type == exceptions.NameError:
            cd = tb.tb_frame.f_locals["code"]
            print cd
      else:
            # follow this to catch all other errors!
            code.InteractiveInterpreter.showtraceback(self)
```

What this does is catch the exception. If the exception is a NameError, it grabs the code that caused the exception and prints it out. If it is not a NameError, it continues with the traceback as normal.

But why do this? My application imported data from databases and the names of columns / fields were not always valid Python variable names (sometimes including white space, starting with numerics and so on). It would have been possible to change the names and tell our users that they have to be more sensible with naming. The other alternative (which might get purists a bit annoyed!) was to use the name strings as variables.

This was done by watching our for attribute errors, catching them, and redirecting them to the object that holds the variables (in this case, as a list in the dataObject we dealt with earlier):

When doing this, remember to use the 'code.InteractiveInterpreter.showtraceback(self)' so that all other errors will be shown.

Other Things

It is possible to build a restricted Python interpreter. In this case, you would simply compare the self.cmd of the STC against the list of keywords that are allowed so that only the valid ones are put through to the interpreter. Of course, it is entirely possible that a clever user will find a way around this so it's not a secure solution. However, it should work for most users if the aim is to reduce complexity.

The Scintilla component is extremely rich and a lot of options can be configured (so many that there could be another article just for them). Things like tooltips, auto-completion, line numbering, and syntax highlighting are all available.

Internationalisation

If your application is internationalised, you have a little thing to watch out for. That little thing is the underscore character, which tells the gettext module to treat the string as one for interpretation. The problem is that the interpreter uses the underscore for something else and this will always overwrite the gettext version. This means that when you set up an interpreter within an internationalised application, you need to substitute the interpreters 'magic' underscore with something else. This function needs to import the sys module (which has already been done) and the __builtin__ module.

```
import sys, __builtin__

def newhook(val):
      if val is not None:
      __builtin__.__last__ = val
            sys.stdout.write('%r\n'%val)
```

Then put this code somewhere before the interpreter is instantiated:

```
sys.displayhook = newhook
```

This code came courtesy of Peter Otten from the comp.lang.python newsgroup.

Embedding iPython

iPython is an extremely powerful extension and can also be embedded easily into a program. Although a program's variables and objects can all be examined, this is a one-way process and any changes are not propagated back to the program.

```
from IPython.Shell import IPShellEmbed

ipshell = IPShellEmbed()

ipshell() # this call anywhere in your program will start IPython
```

Conclusion.

You can see that more programming is required for the InteractiveInterpreter class than the InteractiveConsole class, but it offers a lot more power. Most of the code for the InteractiveInterpreter class concerns the user interface like doing things like indentation properly. But once you have it in place, you can easily build an interpreter that would fit into almost any application. The biggest danger is that because it's so useful, you may be tempted to put an interactive interpreter in every program that you do!

aReference:
Otten, Peter[__peter__@web.de] “gettext and the interpreter” In [comp.lang.python] 4 April 2004.

Appendix
This is all the code together

```
import wx
import wx.stc as stc
import code
import sys
import __main__


#x = code.InteractiveConsole()
#x.interact("A kind of Python interpreter")


class II(code.InteractiveInterpreter):
      def __init__(self, locals):
            code.InteractiveInterpreter.__init__(self, locals)

      def Runit(self, cmd):
            code.InteractiveInterpreter.runsource(self, cmd)

class PySTC(stc.StyledTextCtrl):
      def __init__(self, parent, ID, pos=(10,10), size=(700, 600),
style=0):
            stc.StyledTextCtrl.__init__(self, parent, ID, pos, size,
style)
            sys.stdout = self
            sys.stderr = self
            self.Bind(wx.EVT_KEY_DOWN, self.OnKeyPressed)
            self.cmd = ''
            self.lastpos = self.GetCurrentPos()

      def SetInter(self, interpreter):
            self.inter = interpreter

      def write(self, ln):
            self.AppendTextUTF8('%s'%str(ln))
            self.GotoLine(self.GetLineCount())

      def OnKeyPressed(self, event):
            self.changed = True # records what's been typed in
            char = event.GetKeyCode() # get code of keypress
            if (self.GetCurrentPos() < self.lastpos) and (char <314)
or (char > 317):
                  pass
                  # need to check for arrow keys in this
            elif char == 13:
                  """
                  What to do if <enter> is pressed? It depends if
there are enough
                  instructions
                  """
                  lnno = self.GetCurrentLine()
```

```
                  ln = self.GetLine(lnno)
                  self.cmd = self.cmd + ln + '\r\n'
                  self.NewLine()
                  self.tabs = ln.count('\t') #9
                  if (ln.strip() == '') or ((self.tabs < 1) and (':'
not in ln)):
                        # record command in command list
                        self.cmd = self.cmd.replace('\r\n','\n')
                        # run command now
                        self.inter.Runit(self.cmd)
                        self.cmd = ''
                        self.lastpos = self.GetCurrentPos()
                  else:
                        if ':' in ln:
                              self.tabs = self.tabs + 1
                        self.AppendText('\t' * self.tabs)
                        # change cursor position now
                        p = self.GetLineIndentPosition(lnno + 1)
                        self.GotoPos(p)
            else:
                  event.Skip() # ensure keypress is shown

class ecpintframe(wx.Frame):
      def __init__(self, *args, **kwds):
            kwds["size"] = (700,600)
            wx.Frame.__init__(self, *args, **kwds)
            self.ed = PySTC(self, -1)

if __name__ == '__main__':
      Ecpint = wx.PySimpleApp(0)
      I = II(None)
      win = ecpintframe(None, -1, "EcPint - Interactive intepreter")
      win.Show()
      win.ed.SetInter(I)
      Ecpint.MainLoop()
```

The Python Papers, Volume 3, Number 2 (2008)

Designing Semiconductor Heterostructures with Python

ANITTA THOMAS[†]
School of Computing, University of South Africa, P.O. Box 392, Pretoria 0003, South Africa

and

ANDRÉ E. BOTHA
Department of Physics, University of South Africa, P.O. Box 392, Pretoria 0003, South Africa

ABSTRACT

Matplotlib is used to visualize and design electronic potentials in layered semiconductor devices (heterostructures).

1. INTRODUCTION

Quantum mechanics plays a key role in the design of semiconductor heterostructures. The quantum mechanical behaviour of small particles is often very unintuitive and hence, the ability to visualise the data is advantageous, not only as a modelling tool for the design of heterostructures, but also as a teaching aid in quantum mechanical classes. In this article, we present a proof-of-concept application to demonstrate how Python can be used in the design of semiconductor heterostructures. This application provides a designer with the ability to predict how electrons are likely to behave within a semiconductor device without developing prototypes in a laboratory.

The following discussion introduces the quantum theory underlying the application. Although incomplete, it should be enough to provide the reader with an intuitive understanding of what the various quantities represent in practical terms.

2. PHYSICS BASICS

2.1 Schrödinger's Equation

The Schrödinger wave equation of quantum mechanics can be used to model a wide variety of microscopic phenomena to a very high degree of accuracy. For example, the behaviour of nuclei, atomic and molecular systems, liquids, gases and plasmas, can all in principle be calculated by solving the appropriate Schrödinger equation.

For the purposes of this work, which aims to model the behaviour of electrons in semiconductor heterostructures (i.e. layers of different semiconductors grown together), the appropriate equation is the one-dimensional, time-independent, Schrödinger equation:

$$\frac{\mathrm{d}^2}{\mathrm{d}x^2}\psi(x)+V(x)\psi(x)=E\psi(x) \qquad [1]$$

A physically acceptable solution $\psi(x)$ (i.e. the so-called wave function in Eq. [1]) contains all the mechanical information about the electron, but strictly in a probabilistic sense! For example, the probability of finding the

[†] Corresponding author. E-mail: *thomaa@unisa.ac.za*

electron between position x and $x + dx$ is given exactly by $|\psi(x)|^2 dx$. Here the quantity $dx > 0$ represents a very small interval along the x-axis and $|\psi(x)|$ denotes the magnitude of the complex valued wave function, $\psi(x)$.

In Eq. [1], E is the total energy of the electron. The first term on the left represents the kinetic energy of the electron, while the second term represents its potential energy, $V(x)$. The potential energy of the electron is determined by the properties of the semiconductor in which the electron resides. It therefore changes abruptly as the electron moves, within the heterostructure, from one layer to another. This behaviour of the potential is shown in Fig. 1, in which a typical potential V, has been plotted as a function of position x, along the growth direction of the heterostructure. Each abrupt change in this potential corresponds to a transition from one semiconductor layer to the next. In Fig. 1, for example, there are seven layers in the heterostructure (not counting the outer two layers, in which the potential is zero).

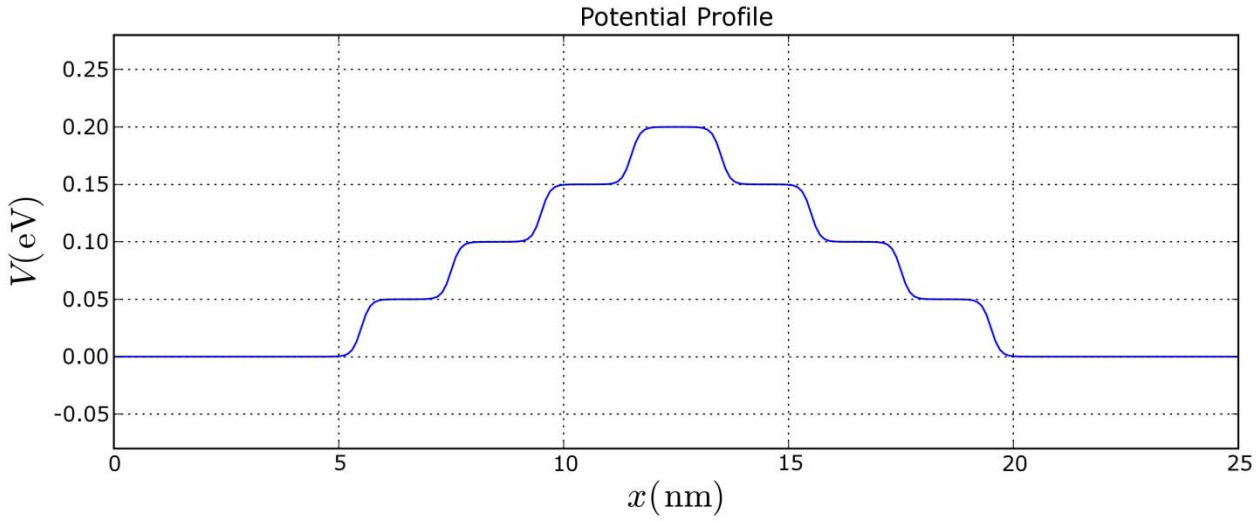


Fig. 1. A typical potential, in units of electronvolt (eV), for a heterostructure lying between 5 and 20 nm (1 nm = 10^{-9} m).

2.2 The Direct Problem

In general we can assume that the heterostructure is situated in some finite interval (x_L, x_R), and that the potential is zero everywhere outside of this interval. We then consider what happens in a so called scattering experiment, in which an electron approaches the interval (called the interaction interval) from the left and either passes through the interval or else gets reflected backwards. Since $V(x) = 0$ to the left of the interaction interval the incident electron has a plane wave, wave function given by e^{ikx}. The reader may easily verify (by substitution) that e^{ikx} is a solution to Eq. [1], provided $k = \pm\sqrt{E}$. Physically the positive sign in the last equation is interpreted as an electron incident from the left. The negative sign corresponds to an electron which has been reflected backwards by the non-zero potential within the interaction interval. In general, when a unit flux of electrons is incident upon the interactive region from the left, a certain percentage of electrons get reflected backwards and the remainder move through the interaction interval and continue indefinitely towards the right.

Mathematically, the wave function outside of the interaction interval thus has the form:

$$\psi(x) = \begin{cases} e^{+ikx} + R(k)e^{-ikx} & \text{, when } (x < x_L) \\ T(k)\,e^{+ikx} & \text{, when } (x_R < x) \end{cases} \qquad [2]$$

The quantities $R(k)$ and $T(k)$ in Eq. [2] are called the reflection and transmission coefficients, respectively. Physically, they are interpreted as follows: $|R(k)|^2$ gives the probability that the electron is reflected backwards, while $|T(k)|^2$ gives the probability that it is transmitted through the interaction interval towards the right. Note that since the electron must either be reflected or else transmitted, the total probability for these two events must be equal to one, i.e. $|R(k)|^2 + |T(k)|^2 = 1$.

If the potential $V(x)$ is known throughout the interaction interval, then both $R(k)$ and $T(k)$ can be calculated by solving the Schrödinger equation numerically, using Eq. [2] as boundary conditions. This problem of calculating $R(k)$ from a given $V(x)$ is referred to as the *direct* problem. As an example, Fig. 2 shows the calculated electron reflectance $|R(k)|^2$ as a function of k, for the potential in Fig. 1. In this case the heterostructure functions as a high-pass filter, effectively blocking all electrons with $k < 0.3$ nm^{-1} and allowing almost complete transmission of electrons with $k > 0.5$ nm^{-1}. Further details on the solution of the direct problem can be found in Ref. (1) and the references therein.

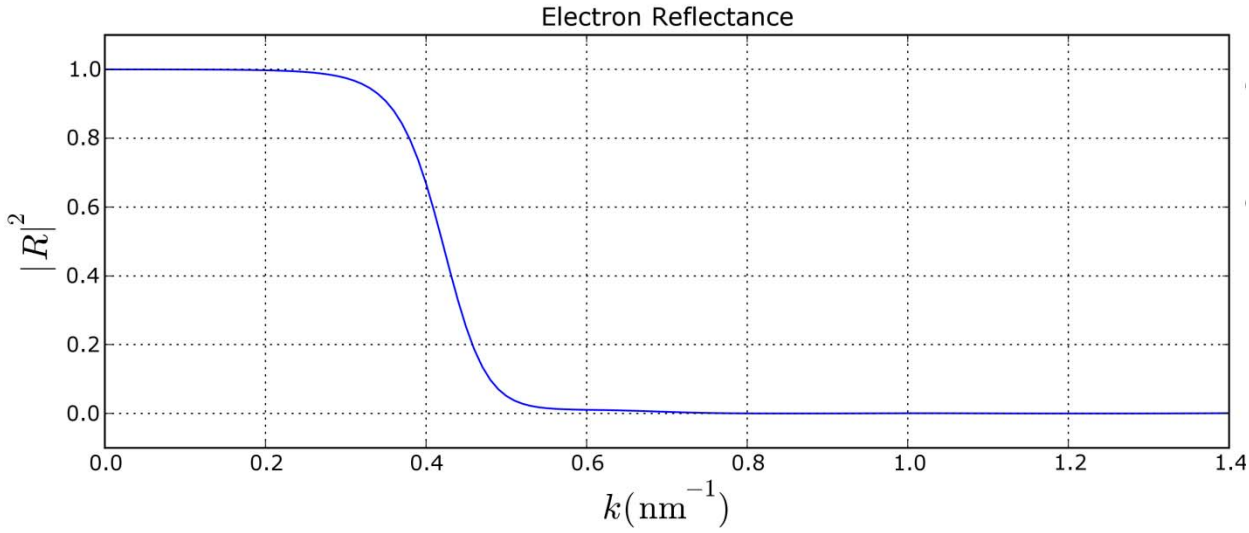


Fig. 2. Calculated electron reflectance as a function of wave vector k, corresponding to the potential in Fig. 1.

2.3 The Inverse Problem

In simple terms, the *inverse* problem consists of recovering $V(x)$ from $R(k)$. Given that a designer may want a device which has a particular effect on electrons, it is natural to ask what should the potential look like? Theoretically the answer to this question is obtained by calculating $V(x)$ from the desired $R(k)$. The calculations for this inverse problem, however, turn out to be substantially more difficult than those for the direct problem. Nevertheless, so-called inverse quantum scattering theory has developed considerably over the past few years and the problem can now be solved. Briefly, it entails solving the Marchenko integral equation:

$$K(x,y) + B(x+y) + \int_{-x}^{+x} \mathrm{d}z\, B(z+y) K(x,z) = 0, \qquad \text{with} \quad y < x \quad , \qquad [3]$$

which takes, as input, the Fourier transform of $R(k)$. In Eq. [3], $B(x)$ denotes the Fourier transform of $R(k)$. If the active interval of the potential lies between $0 < x_L$ and $x_R < x_{max}$, the potential may be recovered from the solution to Eq. [3], i.e. $K(x, y)$ for $x,y \in [0, x_{max}]$ and the relation:

$$V(x) = \frac{\mathrm{d}}{\mathrm{d}x} K(x,x) \qquad [4]$$

For completeness we mention that the phase of reflection coefficient $R(k)$, which is defined as the ratio of the imaginary part of $R(k)$ to the real part of $R(k)$, cannot be measured experimentally. It is because the reflectance $|R(k)|^2$ corresponds more naturally to the real experimental situation, that it is used here to apply the above theory. Further details about the inverse problem can be found in Ref. (2) and the references therein.

3. THE FORTRAN CODES

The present application allows the user to visualize data which is generated by the numerical solution of the quantum mechanical equations described in the previous section. Without Python's ability to make use of codes written in Fortran, the present visualization would not be possible. The equations are simply too complicated. Even with the highly efficient Fortran codes which we use, the calculations take just about as long as one can reasonably expect a user to wait for the curves to be updated – on a 3GHz desktop computer it takes 5 to 8 seconds to perform either the direct or inverse calculation.

A full discussion of the numerical difficulties encountered in solving the aforementioned equations is well beyond the scope of this article. Nevertheless, in order to provide a better understanding of how the Python code functions, we provide the following brief explanation. The module `marchenko.pyd` was created from Fortran 95 codes which were developed and perfected over a period of several years, by using the `f2py` conversion utility available through Numerical Python (`numpy`). Details concerning the use of `f2py` are provided in Ref. (3).

The module `marchenko.pyd` makes available three functions as listed below:

```
>>> import marchenko
>>> print marchenko.Reflectance.__doc__
reflectance - Function signature:
  re = reflectance(vx)
Required arguments:
  vx : input rank-1 array('d') with bounds (nv)
Optional arguments:
  nv := len(vx) input int
Return objects:
  re : rank-1 array('d') with bounds (nq)
```

The function `reflectance()` takes the potential as argument, in the form of an array `vx` of points (x, V), and returns the reflectance in the form of an array `re` of points (x, $|R(k)|^2$, φ), where φ is the phase of the reflection coefficient $R(k)$. As indicated by the above function signature, if N points in the form (x, V) are passed to `reflectance()` then `nv` = 2N and the array `vx` has bounds [0, `nv`-1]. Similarly, if M points in the form (x, $|R|^2$, φ) are desired as output then `nq` = 3M and `re` has bounds [0, `nq`-1]. The potentials in the present work are specified by 500 points, i.e. `nv` = 1000 and `nq` is set (within the Fortran code) to 2250, i.e. 3 times 750. Doubling the value of `nq` (or `nv`, or both `nq` and `nv`) has no visible effect on the calculated reflectance. The Fortran code calculates the reflectance by using the Numerov-Cowling method described in Ref. (4).

```
>>> print marchenko.cubic.__doc__
cubic - Function signature:
  xq,fq = cubic(xp,fp,np,nq)
Required arguments:
  xp : input rank-1 array('d') with bounds (np)
  fp : input rank-1 array('d') with bounds (np)
  np : input int
  nq : input int
```

```
Return objects:
  xq : rank-1 array('d') with bounds (nq)
  fq : rank-1 array('d') with bounds (nq)
```

The function `cubic()` can interpolate smooth curves through a given set of points using the cubic spline interpolation. The use of this function in the present application is discussed later in the article. Details on numerical method of cubic spline interpolation can be found in Ref. (5).

```
>>> print marchenko.inversion.__doc__
inversion - Function signature:
  vv = inversion(re)
Required arguments:
  re : input rank-1 array('d') with bounds (nq)
Return objects:
  vv : rank-1 array('d') with bounds (1000)
```

The function `inversion()`, calculates the potential from the reflectance by solving the Marchenko integral equation [3], using the Nystrom method. A description of this numerical method can be found in Ref. (6).

Next, we will show how Matplotlib is used, in conjunction with the above three functions, to develop an application for designing semiconductor heterostructures. The main advantage of having such an application is that it allows the user to perform both the direct and inverse calculations, which are highly non-trivial, by merely using a computer mouse, and the results of the calculations can be viewed almost immediately.

4. APPLICATION REQUIREMENTS

The data for this application consists of lists of points in the form (x, y), where each list represents either a curve of the potential profile (x, V) or electron reflectance $(k, |R|^2)$ (for simplicity, we will suppress the phase in this part of the discussion). Since it is possible to calculate the potential from the reflectance and vice versa, each curve has a corresponding partner. The user interface therefore consists of two windows, one for each type of curve. Several curves of a particular type may be represented within one window at any given time, with their corresponding partners in the other window. The application should then provide the user with the ability to:

(1) Choose any curve for modification
(2) Select a specific interval of the chosen curve for modification
(3) Specify new points within the selected interval
(4) Construct a smooth curve (using the `cubic()` function written in Fortran) through the selected points
(5) View the modified curve
(6) Update its corresponding partner curve in the other window
(7) Delete unwanted curves (their partners should be deleted automatically)

The following generic features were added to:

a) Provide a way to select the processes (1) to (7) above
b) Present the user with a choice of several starting potential profiles
c) Limit the maximum number of curves in one window to three
d) Start the application with a default potential profile

5. IMPLEMENTATION DETAILS

5.1 User Interface Basics

The two windows in the application support similar functionalities and hence they have similar layouts. Both windows are therefore created by using a single function `initialiseFigure()`, which specifies all their common features.

```
1.   def initialiseFigure(xlabel, ylabel, title):
2.       """
3.       Create a figure.
4.
5.       xlabel and ylabel refer to the x and y axes labels.
6.       title refers to the title of the figure.
7.       Return handles to the created figure and subplot.
8.       """
9.       fig = figure(figsize =(12,4.5))
10.      ax = fig.add_subplot(111)
11.      subplots_adjust(left=0.07,right = 0.8,bottom = 0.2)
12.      ax.set_xlabel(xlabel, size=19)
13.      ax.set_ylabel(ylabel, size=19)
14.      ax.set_title(title)
15.      fig.text(.85,0.86, 'KEY SELECTION', fontweight='bold')
16.      fig.text(.81,0.79, 'Select Graph', fontsize='13')
17.      fig.text(.81,0.75, '"r"--red', fontweight='13')
18.      fig.text(.81,0.71, '"b"--blue', fontweight='13')
19.      fig.text(.81,0.67, '"g"--green', fontweight='13')
20.      fig.text(.81,0.60, 'Select Process', fontsize='13')
21.      fig.text(.81,0.55, '"t"',fontsize = '13')
22.      fig.text(.84,0.55, '-- 2 points')
23.      fig.text(.81,0.51, '"p"',fontsize = '13')
24.      fig.text(.84,0.51, '-- new points')
25.      fig.text(.81,0.47, '"i"',fontsize = '13')
26.      fig.text(.84,0.47, '-- interpolate')
27.      fig.text(.81,0.43, '"d"',fontsize ='13')
28.      fig.text(.84,0.43, '-- delete')
29.      fig.text(.81,0.39, '"u"',fontsize = '13')
30.      fig.text(.84,0.39, '-- update')
31.      return fig, ax
```

Lines `9-14` create a `figure` (window) with the given size; add a canvas for plotting curves, and set the appropriate axis labels and title for the figure. Lines `15-30` enable the generic feature (a) mentioned at end of the previous section (Application Requirements). Line `31` returns the handles for the created `figure` and `subplot`, which are used later in the application to change the properties of the figure and the subplot.

The window that displays potential profiles has an additional set of radio buttons to support the generic feature (b), also mentioned at the end of the previous section. The radio buttons are added by a simple user-defined function named `addRadioButtons()`, which is given below:

```
1:    def addRadioButtons(fig):
2:        """
3:        Add radio buttons to the figure.
4:        """
5:        fig.text(0.81, 0.33, 'Select Potential', fontsize='13')
6:        rax = axes([0.83, 0.10, 0.08, 0.20],axisbg='0.75')
7:        radio = RadioButtons(rax, ('--1', '--2', '--3', '--4', '--5','—
8:        6','--7'), active=4)
```

Using the functions `initialiseFigure()` and `addRadioButtons()` the two windows are created by the following statements:

```
figure1, axis1 = initialiseFigure
(r'$x\/\rm{(nm)}$',r'$V\/\rm{(eV)}$','Potential Profile')
addRadioButtons(figure1)
figure2, axis2 = initialiseFigure
(r'$k\/\rm{(nm}^{-1}\/\rm{)}$',r'$\mid R \/ \mid^{2\/}$','Electron
Reflectance')
```

In the above listing, `figure1` and `figure2` refer to the windows for the potential profile and electron reflectance, respectively (See top left hand corner of Figures 3 and 4, for example).

5.2 The `Graph` Class

A curve is modelled using the class `Graph` - a template for any valid curve in the application. The `Graph` class has attributes to specify the curve in terms of a list of (*x*, *y*) values, its unique colour and its type, i.e. either potential profile or electron reflectance. The constructor of the class is given below:

```
1.   def __init__(self, x, y, color, type):
2.       """
3.       Define a Graph class.
4.
5.       x and y refer to the (x,y) values of the curve.
6.       color refers to the color in which the curve is represented.
7.       type is a string to describe the type of the curve i.e. potential
8.       profile or electron reflectance.
9.       """
10.      self.x = x
11.      self.y = y
12.      self.color =color
13.      self.type = type
```

5.3 Visualising Multiple `Graph` Objects

Multiple `Graph` objects are handled in the application using a `list` variable `listGraphObjects`. To visualise multiple `Graph` instances two aspects should be considered; (i) to draw curves in the respective windows and, (ii) to set the axis limits of the `subplot`s.

The drawing of curves is achieved using the function `updatePlots()`, which is listed below:

```
1.   def updatePlots(ty):
2.       """
3.       Update the curves in a figure.
4.
5.       The input parameter ty is used to determine which figure should be
6.       updated.
7.       """
8.       temp = [x for x in listGraphObjects if x.typ == ty]
9.       Xmin, Xmax, Ymin, Ymax = findLimits(ty,temp)
10.      if(ty == 'profile'):
11.         for o in temp:
12.           axis1.plot(o.x, o.y,(o.color[0]).lower())
13.         axis1.set_xlim(Xmin, Xmax)
14.         axis1.set_ylim(Ymin, Ymax)
15.         axis1.grid(True)
16.         figure1.canvas.draw()
17.      elif(ty == 'reflectance'):
18.         for o in temp:
19.           axis2.plot(o.x, o.y,(o.color[0]).lower())
20.         axis2.set_xlim(Xmin, Xmax)
21.         axis2.set_ylim(Ymin, Ymax)
22.         axis2.grid(True)
23.         figure2.canvas.draw()
```

This function takes an input parameter `ty` to determine whether the window for visualising curves is `figure1` or `figure2`. Using this input parameter, the function creates a list `temp` of all the `Graph` objects of type `ty`, calculates the *x* and *y* limits of all the `Graph` objects in `temp` using the function `findLimits()`(discussed below), plots the curves and sets the limits of the axes in the subplots.

Usually the limits of the axes in a subplot are determined by using the (*x*, *y*) values of the relevant `Graph` objects and the predetermined offset values, if required. However, in this application, both the *x* and *y* limits of the `subplot` for visualising electron reflectance are fixed by experimental considerations. Similarly the *x* limits of the subplot for visualising the potential profile are also fixed by the spatial extent of the heterostructure.

Based on these assumptions, the function `findLimits()` is implemented as:

```
1.   def findLimits(ty, subsetList):
2.       """
3.       Determine the maximum and minimum of x and y values of the
4.       Graph objects.
5.
6.       subsetList is the list of Graph objects for which the x and y
7.       limits should be calculated.
8.       ty refers to the type (electron reflectance or potential profile)
9.       of the curve.
10.      Return the maximum and minimum of x and y values of the given Graph
11.      objects.
12.      """
13.      Ymin = -0.1
14.      Ymax = +1.1
```

```
15.      Xmin = 0.0
16.      Xmax = 1.40001
17.      if(ty == 'profile'):
18.        Xmax = 25.0
19.        Xmin = 0.0
20.        if(len(subsetList)==0):
21.          Ymin = 0.0
22.          Ymax = 0.4
23.        else:
24.          aMinY = min(subsetList[0].y)
25.          aMaxY = max(subsetList[0].y)
26.          for o in subsetList:
27.            aMinY = min(aMinY,min(o.y))
28.            aMaxY = max(aMaxY,max(o.y))
29.          Ymin = aMinY - abs((aMaxY - aMinY)/2.5)
30.          Ymax = aMaxY + abs((aMaxY - aMinY)/2.5)
31.      return Xmin, Xmax, Ymin, Ymax
```

Lines `17-19` are used to set the *y* limits of the subplot in `figure1` to predetermined values in the event of an empty `listGraphObjects` list. This scenario occurs when the user deletes all the curves in `figure1`. Deleting all the curves in `figure2` does not require any additional coding since the *x* and *y* limits are fixed.

Lines `24-28` determine the minimum and maximum *y* values of all the `Graph` objects in the `subsetList`. Appropriate offsets are added to these values by lines `29` and `30`.

5.4 Creating Potential Profile and Electron Reflectance Curves

The user is given a number of initial potential profiles, which are essentially files containing (*x*, *y*) values. The application starts with a default potential profile file, thereby creating a `Graph` object for the respective potential profile curve. It also calculates the electron reflectance and creates the corresponding `Graph` object.

A `Graph` object is created in the application using a function `createGraphObject(),` which implements the following:

- It checks whether the creation of a new `Graph` object satisfies the criteria of not having more than three curves in a window.
- If the criteria above is satisfied, the function assigns a colour to the curve, which is not already taken by the existing curves in the window.
- It creates a new `Graph` object, with the assigned colour and (*x*, *y*) values given to the function.

After initialising the windows, the following statements create a `Graph` object and the corresponding curve in `figure1`.

```
f1 = load('potential5.dat')
createGraphObject(f1,'profile')
updatePlots('profile')
```

The curve displayed, for example, is shown in Figure 3.

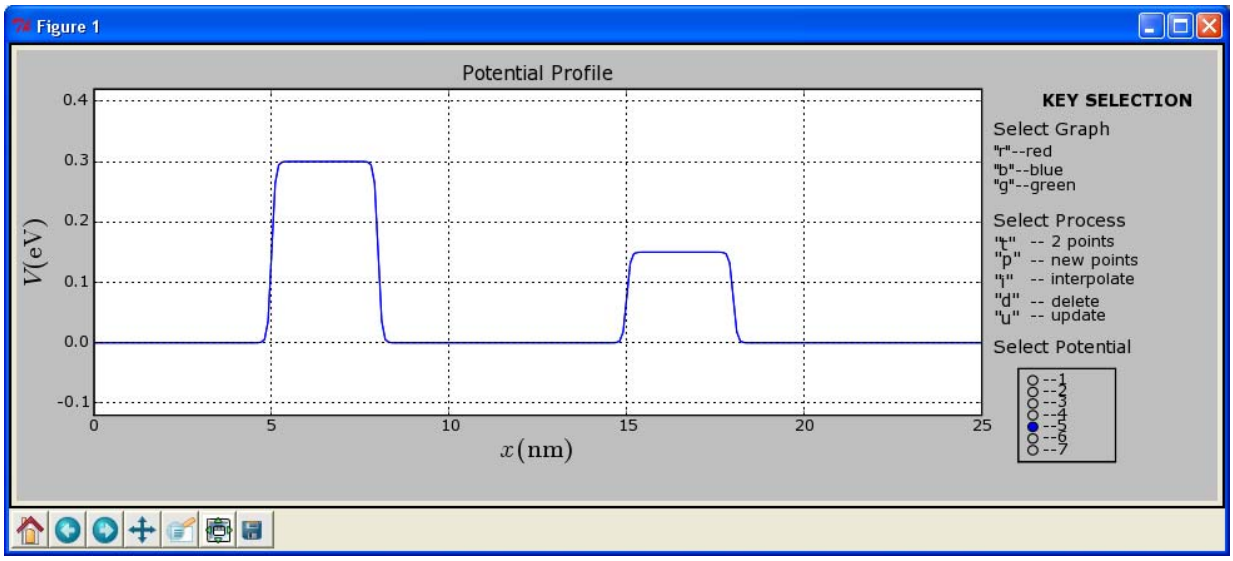


Fig. 3. The display window for the potential. In this simple example the heterostructure consists of three layers which are located between 5 and 17 nm. Notice that the central layer of this heterostructure is made of the same material as the surrounding substrate material.

The corresponding partner for this curve is calculated and created using a function named `electronReflectance()`, which performs the following:

- It invokes the Fortran code `reflectance()`, with the (*x*, *y*) values of the potential profile curve to obtain the electron reflectance data.
- Using the (*x*, *y*) values returned by `reflectance()` a new `Graph` object (of type '`reflectance`' is created).
- `updatePlots()` is invoked to draw the new curve in `figure2`.

After a potential curve has been created, the statement `electronReflectance(f1)` thus creates the partner curve of the potential, as shown in Fig. 4.

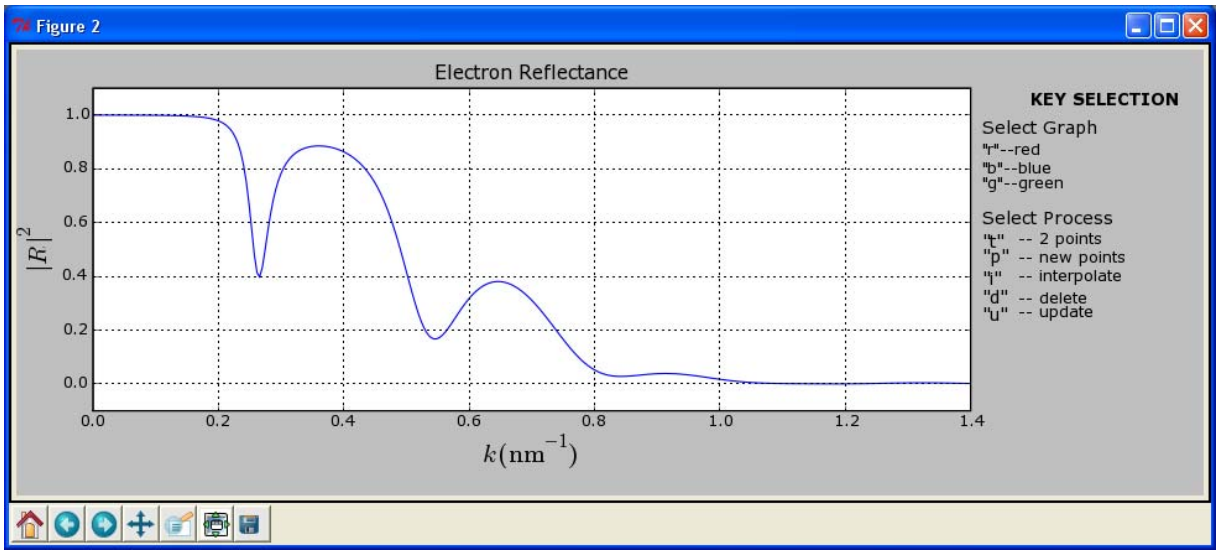


Fig. 4. The display for the electron reflectance corresponding to the potential in Fig. 3.

Any time when a new curve of electron reflectance has to be created, the function `electronReflectance()` is invoked. In a similar way, there is another function `potentialProfile()` which creates a curve of the potential profile, when given the electron reflectance. The function `potentialProfile()` works similarly to `electronReflectance()`, except that it invokes the Fortran code `inversion()` to calculate the (*x*, *y*) values of the potential profile.

5.5 Event Handling in the Application

The application handles key and mouse events to achieve interactivity. Key events allow the user to select a curve for modification, to choose an interval for modification on the selected curve, to specify new points within the chosen interval, to create a new curve through the specified points, to create the partner curve and to delete any curve (and its partner). The choice of an interval for modification, the specification of new points within the chosen interval and the selection of a potential profile from a list of profiles are achieved using mouse events. Again, the events handled in both windows are almost the same; the only difference being the Fortran functions which are invoked to perform either the direct or inverse calculation.

A figure in Matplotlib is registered for key events using the statement:

```
figure.canvas.mpl_connect('key_press_event', delegateProcess)
```

In this application, the function linked to the `key_press_event`, invokes appropriate methods based on the keys entered and manages connection ids for mouse events. A basic layout of the method `delegateProcess` is given below:

```
1:def delegateProcess(event):
2:    if((event.key == 'r')or( event.key == 'b')or(event.key =='g')):
3:       # Choose a curve
4:   if event.key == 't':
5:       # Allow the user to specify the interval to be modified
6:    elif event.key == 'p':
7:      # Allow the user to choose points in the selected interval
8:    elif event.key =='i':
9:       # Draw a new curve based on the new points
10:    elif event.key == 'd':
11:       # Delete the curve
12:    elif event.key == 'u':
13:    # Create the partner curve
```

Each of these processes and the process of selecting a potential profile from a list of potential profiles are achieved by one or more dedicated functions, as explained in the following six sections.

5.6 Selecting a Curve for Modification

A curve is selected by keying in the colour (for example *r* for red) of the curve in the appropriate window. This event causes the `Graph` object to be selected as the active object in the application. The code below shows how the active `Graph` object is created:

```
1:    def createActiveObject(ty, key):
2:         """
3:          Select a curve for modification.
4:
```

```
5:          ty refers to the type (electron reflectance or potential profile)
6:          of the curve.
7:          key refers to the color of the curve.
8:          """
9:          activeGObject = [x for x in listGraphObjects if ((x.typ ==
10:         ty)&((((x.color)[0]).lower())==key))];
```

5.7 Choosing an Interval for Modification

Once a curve has been selected, the user specifies the interval which is to be modified by first entering the key *t*, and then choosing an interval with the mouse. In this application, an interval is chosen by clicking on two points, which lie on the curve (within a certain tolerance). The two valid points are represented by a visual object (in this case a `RegularPolyCollection`) on the figures.

The function listed below allows the user to choose two points on the selected curve. It also represents the points using visual cues.

```
1:    def selectTwoPoints (event):
2:        """
3:        Allow users to select two valid points on a curve.
4:        """
5:        temp = []
6:        flag = False
7:        n = len(activeGObject[0].x) – 1
8:        deltax = abs((activeGObject[0].x)[n] - (activeGObject[0].x)[n-1])
9:        deltay = abs(max(activeGObject[0].y) - min(activeGObject[0].y))/60.0
10:
11:       for i in range(len(activeGObject[0].x)):
12:          if (((abs(event.xdata - (activeGObject[0].x)[i]))< deltax) and
13:          ((abs(event.ydata - (activeGObject[0].y)[i]))< deltay)):
14:             temp = (((activeGObject[0].x)[i]), ((activeGObject[0].y)[i]))
15:             flag = True
16:             break

17:       if ((flag==True) & (countclicks <2)):
18:          thetwopoints.append(temp)
19:          if (activeGObject[0].type == 'profile'):
20:             offsets1.append(((thetwopoints[countclicks])[0],
21:             (thetwopoints[countclicks])[1]))
22:             figure1.canvas.draw()
23:          elif(activeGObject[0].type == 'reflectance'):
24:             offsets2.append(((thetwopoints[countclicks])[0],
25:             (thetwopoints[countclicks])[1]))
26:             figure2.canvas.draw()
27:          countclicks = countclicks + 1
```

Lines `11-16` check if a mouse click is a valid point on the curve (considering the tolerance specified in lines `8` and `9`). If the selected point is valid and the user has not yet chosen two points the selected point will be added to a list named `thetwopoints`, to indicate the interval to be modified in the curve. Lines `20` and `24` add the selected two

points into a list of points (either `offsets1` or `offsets2`) that should be displayed by the `RegularPolyCollection` object.

In Fig. 5 the two points, indicating the interval for modification, are shown.

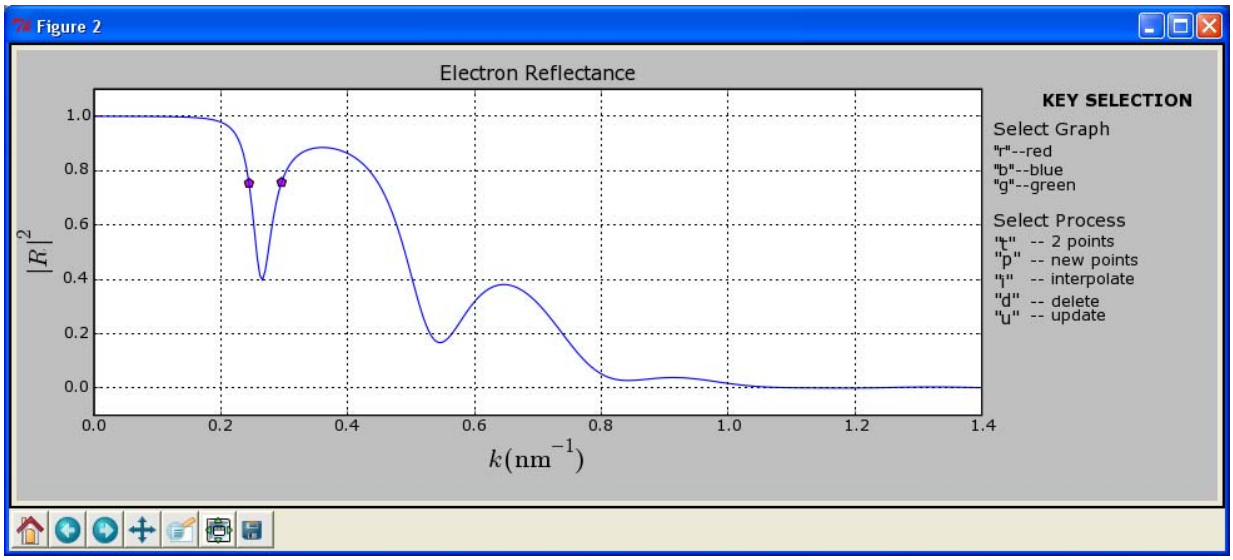


Fig. 5. Electron reflectance, as in Fig. 4, with the two points denoting the interval in which the reflectance is to be modified. In this example the resonance peak between $k = 0.2$ and $k = 0.4$ nm^{-1} will be enhanced to allow almost all electrons with $k = 0.26$ nm^{-1} to pass through the device.

5.8 Specifying the Points within the Chosen Interval

Once the user has chosen the interval for modification, he/she may specify one or more additional points within this interval by keying in *p*. The additional points may then be selected with the mouse. This process should consider two aspects; (i) the user is only allowed to specify points within the chosen interval and, (ii) the user has to be supplied with visual cues representing the specified points.

The function listed below achieves these two requirements:

```
1:    def selectPoints (event):
2:         """
3:         Allow users to select points within the chosen interval.
4:         """
5:          thetwopoints.sort()
6:          if  ((event.xdata < (thetwopoints[1])[0]) & (event.xdata >
7:          (thetwopoints[0])[0])):
8:             selectedpoints.append((event.xdata, event.ydata))
9:             selectedpoints.sort();
10:            if (activeGObject[0].typ == 'profile'):
11:                offsets1.append((event.xdata, event.ydata))
12:                figure1.canvas.draw()
13:            elif(activeGObject[0].typ == 'reflectance'):
14:                 offsets2.append((event.xdata, event.ydata))
15:                 figure2.canvas.draw()
```

As long as the selected points are within the specified interval, they are appended to a `list` named `selectedpoints`, and also to the appropriate lists of the `RegularPolyCollection` objects of the figures.

Fig. 6 shows one additional point selected within the interval of interest.

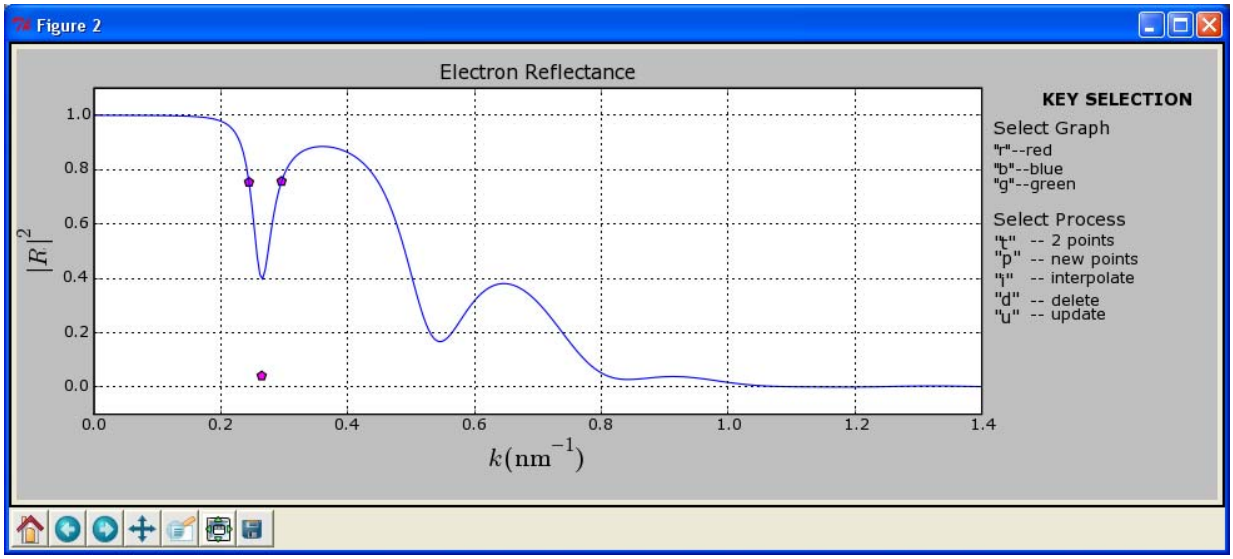


Fig. 6. Electron reflectance, as in Figures. 4 and 5. Here a single point has been chosen between the previously selected two points to enhance the resonance peak at $k = 0.26$ nm^{-1}.

5.9 Creating the Modified Curves

The user types the key *i* to draw a new curve through the specified points. The new curve is a smooth curve which passes through all the specified points and coincides with the unmodified part of the original curve. To achieve this, a function `interpolate()` is defined to implement the following aspects:

- Create a new list of points (*x*, *y*) which includes (*x*, *y*) data of the `activeGObject` by replacing the points in the chosen interval with the new points (`thetwopoints` and `selectedpoints`).
- Invoke the Fortran function `cubic()` to obtain the cubic spline points for the interpolated smooth curve within the modified interval.
- Combine the points (*x*, *y*) in the unmodified section of `activeGObject` with the new points returned by `cubic()` and create a new `Graph` object to represent the new curve.

The modified electron reflectance (green) is displayed in Fig, 7 below.

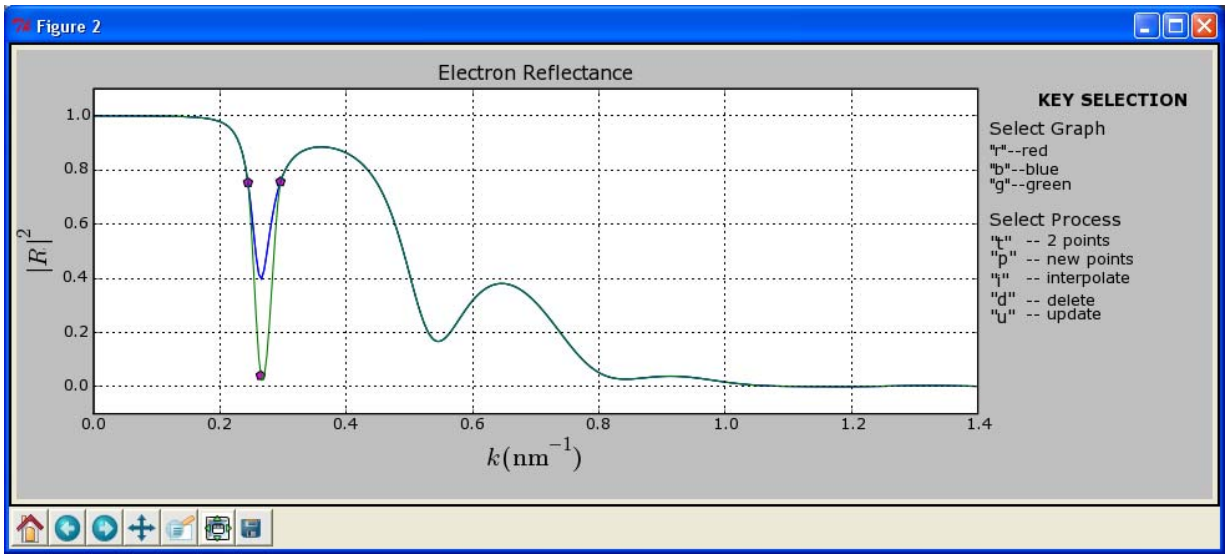


Fig. 7. Electron reflectance, as in Figures 4, 5 and 6. The green curve shows the enhanced resonance peak which has been interpolated through the selected points.

The partner curve (in this case, the potential profile) is created when the user keys in *u*; which invokes, in this case, `potentialProfile()`. The new potential profile corresponding to the modified electron reflectance is shown in Fig. 8.

Fig. 8. The original potential (blue) as well as the modified potential (green) which was calculated from the modified reflectance shown in Fig. 7. This result indicates how the potential should be modified (by, for example, modulation doping) in order to achieve the enhanced resonance peak shown in Fig. 7. Without the inverse calculation it would not be obvious how to dope the heterostructure in order to enhance the resonance.

5.10 Deleting a Curve

An active curve can be deleted by keying *d*, which invokes a user defined function `delete()`. This function simply deletes both `Graph` objects (with the specified colour) from `listGraphObjects`. Subsequently the axes of the subplots in `figure1` and `figure2` are cleared and `updatePlots()` is invoked.

5.11 Choosing Various Potential Profiles

At any time, the user is allowed to select a new starting potential profile by clicking on one of the seven radio buttons. The selection of a new potential involves; deleting all the `Graph` objects in the application, clearing the appropriate lists (e.g. `thetwopoints` and `selectedpoints`), reading the new potential data from the file, creating and displaying the appropriate `Graph` objects.

6. IMPROVEMENTS AND CONCLUSION

Since this application was developed as a proof-of-concept, there is plenty of room for improvement. In terms of the front-end of the application, a more user-friendly interface can be added to make the selection of various options easier. Handling of exceptions, providing feedback to the user on various aspects of the calculations, including features to increase robustness and implementing multi-threading to improve the general efficiency and responsiveness are some of the improvements we intend making in future work.

In terms of the physics underlying this application, several improvements are possible. Firstly, the model can be developed to include bound states (2). Secondly, in order to facilitate more accurate comparisons with experimental data, the model can be extended, either to multi-band $\boldsymbol{k \cdot p}$ models (1), or else by directly using the (periodic) crystal potential (7). Lastly, for designing spintronic devices, which in addition to the change on the electron also exploit its spin degree of freedom, spin-dependence will have to be incorporated into the model.

Our provisional results suggest that further work along these lines would be rewarding. Until now, there has been no commercially available application to provide the user with an interactive visualization of the highly non-trivial numerical computations required for designing heterostructures. Future work, based on refinements of the above ideas, could be of great practical use in the emerging field of computational semiconductor heterostructure design.

ACKNOWLEDGEMENTS

This material is based upon work supported financially by the National Research Foundation of South Africa. The second author (AEB) would like to thank Prof. M. Braun for his helpful (informal) discussions on Python.

BIBLIOGRAPHY

1. **Botha, A. E.** 2007, Microelectronics Journal, Vol. 38, p. 332.

2. **Sofianos S.A., et al.** 2007, Microelectronics Journal, Vol. 38, p. 235.

3. **Langtangen, H.P.** *Python Scripting for Computational Science.* Berlin : Spinger-Verlag, 2004.

4. **Meredith, S.E. and Koonin D.C.** *Computational Physics.* New York : Westview Press, 1990.

5. **De Boor, C.** *A Practical Guide to Splines.* New York : Springer-Verlag, 1978.

6. **Press W.H., et al.** *Numerical Recipes in Fortran.* New York : Cambridge University Press, 1992.

7. **Allen, L.J., et al.** 2001, Acta Crystallographica A, Vol. 57, p. 473.

seismic-py: Reading seismic data with Python

Kurt Schwehr

Center for Coastal and Ocean Mapping, University of New Hampshire

Abstract. The field of seismic exploration of the Earth has changed dramatically over the last half a century. The Society of Exploration Geophysicists (SEG) has worked to create standards to store the vast amounts of seismic data in a way that will be portable across computer architectures. However, it has been impossible to predict the needs of the immense range of seismic data acquisition systems. As a result, vendors have had to bend the rules to accommodate the needs of new instruments and experiment types. For low level access to seismic data, there is need for a standard open source library to allow access to a wide range of vendor data files that can handle all of the variations. A new seismic software package, seismic-py, provides an infrastructure for creating and managing drivers for each particular format. Drivers can be derived from one of the known formats and altered to handle any slight variations. Alternatively drivers can be developed from scratch for formats that are very different from any previously defined format. Python has been the key to making driver development easy and efficient to implement. The goal of seismic-py is to be the base system that will power a wide range of experimentation with seismic data and at the same time provide clear documentation for the historical record of seismic data formats.

INTRODUCTION

Seismic data systems use acoustic pulses to send sound waves through water and the solid earth to map layers within the subsurface. They vary from simple single source and single receiver systems to multiple sources and long arrays of geophones or hydrophones. The processing of the received sound waves requires a range of data storage and signal analysis techniques. Python can support both the data archival and precessing tasks.

In this paper, I will use the example of a single source seismic instrument towed behind a ship (e.g. Figure 1a,b). The device (known as a tow-fish, or just a *fish*) is a 2m long device that emits a pulse of sound energy over a range of frequencies straight down using piezoelectric transducers. The energy travels as waves through the water and bottom material. As the sound velocity of the medium changes, a small portion of the energy is reflected back up towards the fish where it is collected by the receivers and stored for later processing. Each pulse of outgoing energy is referenced to as a *shot* and the resulting returned data are collectively called a *trace*. A GPS on-board the ship records the position and time of the ship for each shot. When traces are combined and georeferenced, a ribbon view is created that is call a seismic line (Figure 1c). The inset in Figure 1c shows the seismic lines combined with the bathymetry to give an overall picture of the ocean bottom. The process of going from shots to a 3D model with interpretation through to a publication can be arduous, especially when there are terabytes data.

There are literally thousands of pieces of code around the world for reading and writing seismic data both in the commercial and academic world. Do we need yet another one? The Society of Exploration Geophysicists (SEG) has worked to provide a number of well thought out standards for seismic data (e.g. SEGY Rev 0) [*Barry et al.*, 1975]. The SEG has continued evaluating the needs of the community and has released an updated format that attempts to accommodate changes in the industry (e.g. SEGY Rev 1) [*Norris and Faichney*, 2002]. However, the reality is that no one software package can read all of the variations on these standard

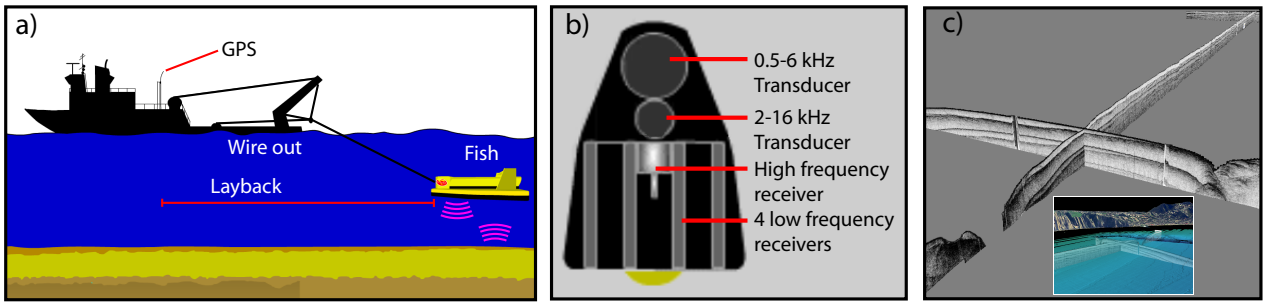


Figure 1. a) Schematic of a ship towing a seismic subbottom profiler (*fish*). Base image courtesy Genevieve Tauxe. b) Underside of an EdgeTech Chirp fish showing the transducers that produce the acoustic *shot* and the sensors that receive the reflected energy, which are stored as a *trace*. Image courtesy Laurent Beguery. c) Traces are shown together as a curtain to show a *seismic line* and are often combined with multibeam bathymetry as visualized by Fledermaus in the inset. Data courtesy Neal Driscoll and Pat Iampietro.

formats and there is no central repository documenting how these formats differ from the SEGY standards. Many seismic processing packages come with tools that allow binary level inspection of data files to attempt to ascertain how a particular SEGY file is structured. Users currently need a range of tools in their arsenal to extract critical data from recorded data streams.

seismic-py is a package designed to alleviate the problems of changing data file formats in the seismic industry. It provides a Python Application Programming Interface (API) for seismic data that relies on a set of drivers that specify and document the actual layout of a particular file format. seismic-py provides this critical functionality in a library released under the GNU General Public License (GPL) [*Stallman*, 1984–], an Open Source Initiative (OSI) [*Raymond*, 1998–] approved license. This means that students and professionals alike have the right to modify and improve the seismic-py package. The source code is available for download here:

`http://schwehr.org/software/seismic-py/`

SEGY FILE LAYOUT

Before proceeding into the details of the software system, it is important to have an understanding of the layout of SEGY Rev 1 data. The format is a Fortran style series of binary data records preceded by a header. The overall layout of SEGY files breaks the content into two major sections. First is a header group starting with a 3200 byte text block (either ASCII or EBCDIC) that is either free form or grouped into 40 predefined 80 character records. Following the text block is a well defined 400 byte binary header region. After this are zero or more Extended Textual File Headers that do not have their format defined in the standard. The rest of the file consists of seismic trace records. These trace records are not required to all be the same size, but they are required to have a 240 byte binary header at the beginning of each trace. Vendors and people processing seismic data frequently create their own format by changing the meaning of these binary fields.

DESIGN

The choice of computer language is the most pivotal design element of a software project. This choice alters which tasks will be hard or easy. For most seismic packages, Fortran and/or C/C++ are the usual choices for implementation. Fortran is the most common language for the geophysical community, but is rather rigid. The C/C++ family of languages provides extreme flexibility, dynamically loadable modules, object oriented design and much more, but at a cost of complexity and frequency of bugs. Python appears to be an excellent compromise between the two groups of languages. Python comes with additional functionality not easily available with either of the other alternatives. Students are able to quickly pickup skills in Python faster than Fortran or C. Python's additional functionality simplified the initial design and implementation of seismic-py.

With the wide range of vendor implementations of SEGY writers, it is critical that SEGY readers be able to easily handle a large number of drivers and allow driver writers to quickly produce the needed changes. With C/C++, this task is possible with dynamically loaded, shared libraries or by parsing specification files,

```
def createDriverName(drvStr):
    '''Make Python filename to load:
    xstar             -> segy_drv_xstar
    drv_xstar         -> segy_drv_xstar
    segy_drv_xstar.py -> segy_drv_xstar
    '''
    if -1 != drvStr.find('.py'):
        drvStr = drvStr[:-3]
    if -1 == drvStr.find('drv_'):
        drvStr = 'segy_drv_'+drvStr
    if -1 == drvStr.find('segy_'):
        drvStr = 'segy_'+drvStr
    return drvStr

def getDriverModule(drvStr='segy_drv_rev1'):
    drivername = createDriverName(drvStr)
    file, pathname, description =
         imp.find_module(drivername)
    drv = imp.load_module(drivername,file,
                          pathname,description)
    return drv
```

Figure 2. The `getDriverModule` function wraps the Python module loaded. By wrapping the standard Python module loader with the `createDriverName` function, seismic-py is able to let the user use shorthand driver names such as "rev1" instead of "`segy_drv_rev1.py`".

but is prone to errors and difficulties with dynamic linkers. Python provides this functionality with the `imp` module [*Python Software Foundation*, 2008b] allowing device loading to be coded in python. The `imp` module provides the components required to create a custom import function in python. seismic-py provides a `getDriverModule` function that wraps the `imp` module allowing the user to specify the driver name in any one of four forms (Figure 2). Python loads a driver module from the users `PYTHONPATH` and returns the `drv` object. All Python code is then able to access the driver data just as it would any other Python module.

A non object-oriented design works just as well and should be more approachable to scientists who may not be familiar with object-oriented design. For most projects of this nature, the obvious choice for a design would be to create a parent class and derive drivers from the parent class or from sibling drivers. With the history of the SEGY formats, a straight inheritance tree would probably be rather difficult. It is expected that as the pool of drivers increases, new drivers will pull pieces

```
1. textFileHeaderEntries (Optional)
2. binaryHeaderEntries
3. extTextFileHeaderEntries (Optional)
4. traceHeader
5. fileHeaderTables
6. traceHeaderTables
7. fileHeaderShortList
8. traceHeaderShortList
```

Figure 3. Lookup tables (dictionaries) for a SEGY file driver. Each table specifies all of the valid field names and byte locations for each field. All of these tables are required except the text and extended text file header entries.

from a wide variety of existing drivers. A true object-oriented design would potentially create a complicated path of multiple inheritances. The driver approach here appears to simplify this problem and allows drivers to reuse pieces from where ever they exist without code duplications. If one were to try to draw the historical relationships of SEGY format variations, it might look something like the attempts to graph Unix system lineages: very complicated and never truly accurate.

Driver Specification File

Each specification driver is simply a Python file with a set of required dictionaries (lookup tables; Figure 3). These lookup tables have a variety of tasks ranging from acting as pointers into binary data to allowing decoding of data elements. Each section dictionary contains byte offset ranges for each data field. The "`segy_drv_rev1.py`" provides the reference driver.

Items 1-4 in Figure 3 provide the core lookup tables. These tables specify the location for each field in the headers. The `binaryHeaderEntries` and `traceHeader` tables together dictate how to decode the data in the traces. The majority of these fields are integers. For those integers that are enumerated values, it is important to be able to create human readable text representations of values. Take the `dataField` dictionary as an example. A '5' means that the data will be a "4-byte IEEE floating-point." The lookup tables provide byte offsets in items 5 and 6 for each field. Figure 4 shows a code example showing how to use the tables contained in a driver.

The short lists (items 7 and 8) are used for programs that wish to show a smaller list of items considered to be the most critical. The short list provides a less overwhelming view of the trace header and are the items

```
>>> import segy
>>> s = segy.Segy('file.sgy')
>>> print s.drv.binaryHeaderEntries
                ['SampleFormat']
    [3225, 3226, 'Data sample format code']
>>> s.header.getBinEntry('SampleFormat')
    1
>>> print s.drv.fileHeaderTables
                ['SampleFormat'][1]
    4-byte IBM floating point
```

Figure 4. The Python command line is a quick way to explore a SEGY file. Once a file is loaded, it is possible to query for the raw values as with getBinEntry or get the English translation by using one of the lookup tables.

```
from segy_drv_rev1 import dataFormats
from segy_drv_rev1 import dataFormatStruct
from segy_drv_rev1 import traceSortingCodes
from segy_drv_rev1 import sweepTypeCodes
```

Figure 5. Reuse of common driver functionality is encouraged. This can also be used to show the heritage of file format. For example, if a driver is essentially SEGY Rev 1 with a few modifications, this will immediately be clear to anyone who reads the driver file.

that the driver author decided are the most important for users to examine. For example, the short list for a trace header might consist of only the shot number (*Shotpoint*), the geographic location of the GPS (X, Y), and the delay from the shot firing to the time the receivers start recording (*Delay*). The standard trace header has an overwhelming 90 items, whereas the short list might have just 4 or 5 entries.

Deriving Variant Specifications

Once a basic driver has been created for a family of SEGY formats, it is easy to create derivative drivers that only modify small portions of an existing driver. The `segy_drv_xstar.py` file provides an example of a derivative driver. The SIO EdgeTech Chirp XStar format is similar to SEGY Rev 1. All of the components that remain the same are directly imported (Figure 5).

Python tries to keep only references to objects when they are used elsewhere within a Python program. For items that need to be changed, it is important to make a completely new and separate local copy of the data. This is done with what Python calls a `deepcopy` [*Python Software Foundation*, 2008a]. Figure 6 is an example with the binaryHeaderEntries. The XStar format does not fill in a number of fields. Missing entries are removed from the local copy after the SEGY Rev 1 entries are deep copied.

```
binaryHeaderEntries = copy.deepcopy (
    segy_drv_rev1.binaryHeaderEntries
    )
del binaryHeaderEntries['JobId']
del binaryHeaderEntries['ReelNo']
del binaryHeaderEntries['TracesPerEnsemble']
```

Figure 6. It is critical to use deepcopy when deriving tables from drivers. This prevents the original driver from being corrupted when altering of deleting entries in a new driver.

Performance

Software performance is critical to seismic processing applications. Seismic instruments are capable of rapidly generating enormous quantities of data. If the code is not able to cope with this volume, users will quickly become frustrated. seismic-py takes the approach of using the mmap system call through the `mmap` Python module [*Python Software Foundation*, 2008c]. This call allows the operating system (OS) to page data into memory on demand via the paging system. Since these pages are marked as read only, the OS can dump pages quickly as memory pressure increases during processing runs. Locations of each name are stored in a Python dictionary (basically hash tables). With the small size of these dictionaries, the lookups proceed quickly. `mmap` brings in raw binary data that cannot be direct read with Python. However, Python provides the `struct` module [*Python Software Foundation*, 2008d] that can convert binary data to Python objects given a conversion string. The `struct` module can convert a range of integer types along with IEEE 32- and 64-bit floating point numbers. Much older seismic data is in IBM floating point format that is not supported by struct, therefore seismic-py can not yet read those seismic data files.

If the speed of the pure Python is not fast enough, it is possible to replace data parsing code with optimized C or C++ code. Originally, this was only possible with the Python/C programming API [*van Rosum*, 2008], but there now exist a wide range of tools for wrapping C++ for using python such as SWIG [*Beazley and Lomdhal*, 1997] or Boost.Python [*Abrahams*, 2002–], or

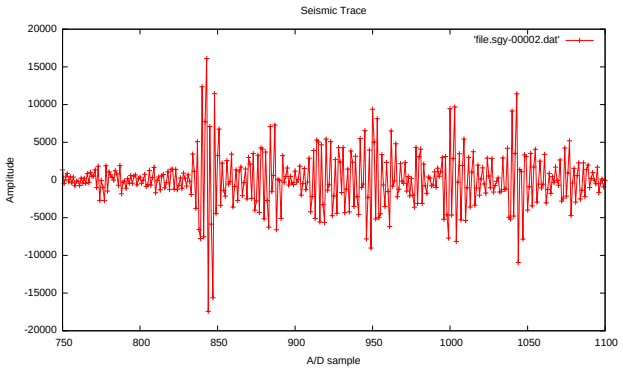


Figure 7. Gnuplot output from plotting the ASCII trace values written to disk by `segydump`. Plotted with "`plot 'file.sgy-00002.dat' with linespoints`".

alternatively using C inline within Python source code [*Mardal and Westlie*, 2007-; *Simpson*, 2001].

SAMPLE APPLICATIONS

To make it easier to get started with seismic-py, the package comes with many sample applications. I will discuss 3 applications to give a flavor of the possibilities. `Segydump` provides a quick look capability similar standard hex viewers, but with an understanding of the header field names. `Seqysql` loads the trace headers into a simple SQL database. `Segysqlgmt` combines the trace locations with a program to draw maps.

Segydump

`Segydump` provides internal listings and trace data dumps for SEGY data files. This is an excellent starting example as it exercises just about all of the functionality in the driver but hides most of it behind the `Segy` class. The most challenging portion of `Segydump` is handling all of the command line options such as being able to include the filename in front of each line of text. Figure 8 is a stripped down version of the dumping code.

By specifying the driver with the `Segy` class, all of the quirks of the XStar format are irrelevant at this level of the interface. The code starts by opening a SEGY data file with the specified driver on Line 3. Line 4 prints out the number of traces in the file. The `printBinaryHeaders` call in line 5 prints out all of the header entries. The user can request that traces be dumped out to disk, which is done in lines 6-8. Line 9 finishes by printing out the header information for each trace. Additional code in `segydump` (not shown here) handles looping over each of the provided files, selecting

```
1 traceNum = 123
2 filename = 'LaJolla-line101.xstar'
3 s = Segy(filename, drivername='xstar')
4 print 'traces = %s' % s.getNumberOfTraces()
5 s.header.printBinaryHeaders()
6 s.writeTraceToFile('%s-%05d.dat' \
7                     % (filename,traceNum),
8                     traceNum)
9 s.getTraceHdr(traceNum).printHeader()
```

Figure 8. This code snippet writes the data from a trace out to an ASCII text data file. This data file is suitable to loading into Octave or plotting with Gnuplot.

short or long output, and providing additional information. It is up to the user to use a tool like grep to pull out specific header fields. Think of `segydump` as the equivalent to the Unix `ls` or DOS `dir` commands.

The ability to write out individual traces should make a wide range of studies more convenient. Most processing environments and languages can read in ASCII data that is in (sample number,value) pairs. The simplest case is visual inspection of individual traces as shown in Figure 7, which shows the `gnuplot` results from running "`segydump -w -t 2 file.sgy`" followed by gnuplot. This idea can be extended to programs such as `MATLAB` and `IDL/ENVI` where additional signal processing is traditionally can be performed.

Segysql

One of the most common tasks of working with seismic data is trying to manage all of the metadata for SEGY files. Python provides a rich set of modules that simplify many of these tasks. Users often want to know which lines cross through a region or the shot closest to a feature, core, or station. The strategy most frequently used in the academic world is to create a wide range of text columns managed with `awk`, `sed`, and Perl scripts. SQL provides an easier way to query large data sets. The problem is that there has been no easy way to import the header data for files and traces into an SQL database. `Segysql` provides a complete example of database importing using the SQLite [*Wyrick and Hipp*, 2000–] database. SQLite was solely for its ease of use. There is no need to setup a database server. As of version 2.5, Python has a SQLite database interface called `sqlite3`, that simply uses a single file as the database repository. Older versions of Python can use pysqlite [*Owens and Haering*, 2001–]. Switching to any

```
01 import segy
02 import sqlite3
03 cx = sqlite3.connect('segy.db3')
04 cu = cx.cursor()
05 cu.execute(segy.sqlCreateFileTable('xstar'))
06 cu.execute(segy.sqlCreateTraceTable('xstar'))
07 cx.commit()
08 xstar = segy.Segy(filename,'xstar')
09 cu.execute(xstar.header.sqlInsert(filename))
10 cx.commit()
11 cu.execute('SELECT fileKey FROM segyFile WHERE filename=:1;',(filename,))
12 fileKey = cu.fetchone()['fileKey']
13 for i in range(1,xstar.getNumberOfTraces()+1):
14     cu.execute(xstar.getTraceHdr(i).sqlInsert(traceNumber=i,fileKey=fileKey))
15 cx.commit()
```

Figure 9. seismic-py provides helper mechanisms to simplify SQL database creation that can easily be combined with SQLite.

other database interface requires changing only a few lines. The seismic-py Python API provides methods that return the necessary SQL string for table creation and row insertion. Just pass these string into a new database interface.

Figure 9 demonstrates a stripped down version of code to create and fill an SQL database from a set of SEGY files. Lines 3-7 connect to a new database file and create the database tables. Once, the database has been created, the first task is to add a file to the database (Line 09). To insert all of the trace headers into the headers, first we have to get (with an SQL `SELECT`) the reference key created by the database for the file (Line 9-10). Finally, each trace is added in a loop over all of the traces (Line 13-14). The `commit` calls are part of the SQL database interface transaction handling. Nothing is actually added to the database until the `commit` call.

Segysqlgmt

Once header information is in a database, it becomes much easier to create mini-applications that add to the seismic processor's tool chest. Marine scientists typically use GMT [*Wessel and Smith*, 2006] and MBSystem [*Caress and Chayes*, 2001–] to make maps of areas that can incorporate other critical data. `segysqlgmt`, a program to display the tracks of seismic lines on a map, illustrates this concept. `mbm_grdplot`, a script with in MB System, reads a GMT `grd` and then outputs a default plotting script using GMT commands, providing a simple basemap. For example, with a Santa Barbara Basin, CA mutlibeam data set [*Hatcher and Maher*, 1999], the command is "`mbm_grdplot -Igmt.grd`." This is much easier that starting off writing your own GMT script. `Segysqlgmt` can then provide text format data files for the ship tracks and shot counts at intervals in a format suitable for GMT's `psxy` and `pstext` along with the shell script lines to add to the `mbm_grdplot` original script. `segysqlgmt` creates quick look shot plots for surveys on top of that base map that can be seen in Figure 10.

FUTURE DIRECTIONS

There is still much work to be done on seismic-py. This paper describes only the initial work done by one developer. seismic-py takes a different approach to seismic data processing compated to other academic packages such as sioseis [*Henkart*, 1975]), Seismic Unix [*Stockwell*, 1997], or pltsegy [*Harding*, 2005] by providing stand alone base level drivers. An open source contribution to the seismic community will hopefully spur more research into seismic data processing, visualization, and interpretation that will give the geoscience community new views into our rocky planets.

To date, seismic-py only implements the SEGY Rev 1 and the EdgeTech XStar format, but it holds promise for providing a vast range of data formats. Critical missing features include full handling of ASCII/EBCDIC headers, extended text headers, IBM floating point data, definition of non-integer header values, and many more vendors' formats. All of these are not hard to provide and are just a matter of additional developer

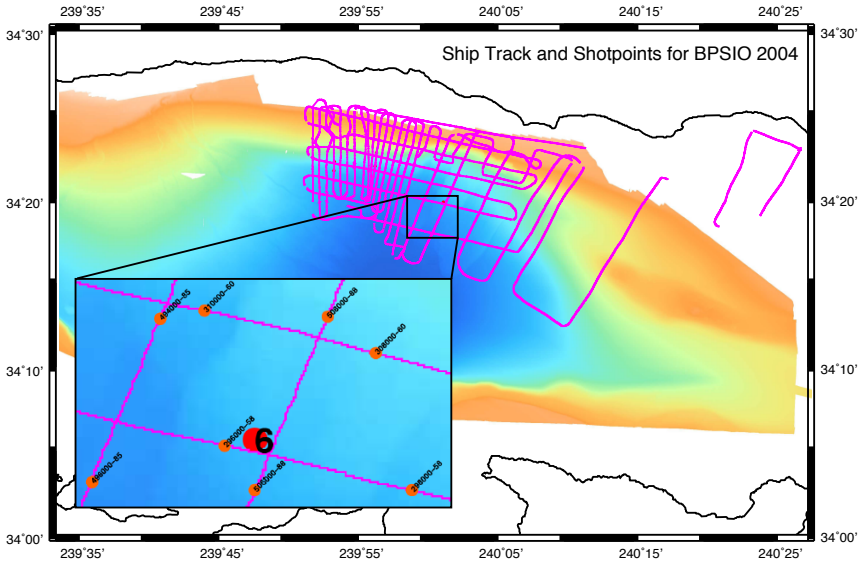


Figure 10. Ship tracks and shot points for a Santa Barbara Basin chirp cruise in 2004. The map was created with a combination of MB-System, GMT, and seismic-py. By drawing cores, ship tracks, and shot numbers, analysts can quickly find the relevant data.

time. Python has proved to be an ideal language for handling formats like this that are Fortran style binary data records. The built-in dictionary and list data types make the driver files appear very close to the original text specification documents.

The `segy` class interface needs a few additions. The most critical to making seismic-py more "Pythonic" is to add an iterator interface, such that a for loop on a `segy` object will loop over the traces. Additionally, the initial database interface only supports traditional SQL calls. Future versions need to add support for spatial databases such as PostGIS [*Refractions Research*, 2008] and [*Furieri*, 2008]

The initial work on seismic-py used Python dictionaries to define the SEGY file format and for the files variances produced by each instrument. This is effective for initial prototypes, but for larger impact on the community, future projects should use eXtensible Markup Language (XML) configuration files. XML allows software implementers to choose their language of preference (assuming that it has an XML reading library) or a compiler could be generated that emits source code for any particular programming language. A compiled version (resulting in Python code) could be made faster without the current run-time table lookups.

CONCLUSION

seismic-py provides a reference implementation of an interface to the wide variety of seismic data that end users encounter in processing seismic data. seismic-py will provide the basis for the rapid development of new tools for inspecting and processing seismic data. seismic-py removes the restriction of using one type of seismic data at a time. Users can open many different seismic streams at the same time with each stream utilizing a different driver matching each data file type.

Today, many students in geophysics and geology learn programming on non-geoscience type problems. Tools like seismic-py will allow beginning students learn the computer languages such as Python while working with data sets that are exciting and cutting edge. Why not have students start by accessing and viewing seismic data when they are learning to program if they are studying geophysics? The hope is to turn seismic-py into one component of an introduction to scientific computing class. The class can use packages such as SciPy [*Jones et al.*, 2001–], ScientificPython[*Hinsen*, 1999–], the pygsl interface to Gnu Scientific Library [*Gaedke and Schnizer*, 2001–]), and the Python Imaging Library [*PythonWare*, 2006] to dive right into processing real data while learning data structures and algorithms. The various tasks can then be combined to create figures

suitable for scientific publications (e.g. Figure 1c).

Perhaps the most important side effect of seismic-py is the beginning of a library that documents seismic formats. There is a vast wealth of commercial and academic seismic data already collected to date. By keeping these older data sets readable, new experiments studying change in earth structures become more manageable. Huge amounts of money have already been spent collecting seismic data and it is important to simplify access to the valuable resource.

Acknowledgments

I would like to thank the reviewers and many other people who read drafts of this paper for their valuable input. Lisa Tauxe and Neal Driscoll provided much encouragement. NSF and BP provided funding the data collection that led to this software project.

References

Abrahams, D., Boost.Python, *http://www.boost.org/doc/libs/release/libs/python/doc/*, 2002–.

Barry, K. M., D. A. Cavers, and C. W. Kneale, Report on recommended standards for digital tape formats, *Geophysics*, *40*, 344–352, 1975.

Beazley, D. M., and P. S. Lomdhal, Feeding a Large–scale Physics Application to Python, *International Python Conference*, *6*, http://www.swig.org/papers/Py97/beazley.html, 1997.

Caress, D., and D. Chayes, MB-System Version 5, Open source software distributed from the MBARI and L-DEO web sites, 12 beta releases, *http://www.ldeo.columbia.edu/res/pi/MB-System/MB-System.intro.html*, 2001–.

Furieri, A., SpatiaLite - VirtualShape, *http://www.gaia-gis.it/spatialite-2.0/index.html*, 2008.

Gaedke, A., and P. Schnizer, PyGSL: Python interface for GNU Scientific Library, *http://pygsl.sourceforge.net/*, 2001–.

Harding, A., pltsegy, 2005.

Hatcher, G., and N. Maher, MBARI Santa Barbara Basin Multibeam Survey, *http://www.mbari.org/data/mapping/SBBasin/default.htm*, 1999.

Henkart, P., Sioseis, *http://sioseis.ucsd.edu/*, 1975.

Hinsen, K., ScientificPython, *http://starship.python.net/~hinsen/ScientificPython/*, 1999–.

Jones, E., T. Oliphant, P. Peterson, et al., SciPy: Open source scientific tools for Python, *http://www.scipy.org/*, 2001–.

Mardal, K., and M. Westlie, Instant, *http://www.fenics.org/wiki/Instant*, 2007-.

Norris, M. W., and A. K. Faichney, SEG Y rev 1 Data Exchange format, *http://seg.org/publications/tech-stand/*, 2002.

Owens, M., and G. Haering, pysqlite - A DB API v2.0 compatible interface to SQLite, *http://initd.org/tracker/pysqlite*, 2001–.

Python Software Foundation, copy – Shallow and deep copy operations, *http://docs.python.org/lib/module-copy.html*, 2008a.

Python Software Foundation, imp – Access the import internals, *http://docs.python.org/lib/module-imp.html*, 2008b.

Python Software Foundation, mmap – Memory-mapped file support, *http://docs.python.org/lib/module-mmap.html*, 2008c.

Python Software Foundation, struct – Interpret strings as packed binary data, *http://docs.python.org/lib/module-struct.html*, 2008d.

PythonWare, Python Imaging Library (PIL), *http://www.pythonware.com/products/pil/*, 2006.

Raymond, E. S., The Open Source Initiative, *http://www.opensource.org/*, 1998–.

Refractions Research, PostGIS, *http://postgis.refractions.net/*, 2008.

Simpson, K., PyInline, *http://www.fenics.org/wiki/Instant*, 2001.

Stallman, R., The GNU General Public License, *http://www.gnu.org/licenses/licenses.html*, 1984–.

Stockwell, J. W., Free Software in Education: A case study of CWP/SU: Seismic Un*x, *The Leading Edge*, 1997.

van Rosum, G., Python/C API Reference, *http://docs.python.org/api/api.html*, 2008.

Wessel, P., and W. H. F. Smith, Generic Mapping Tools, *http://gmt.soest.hawaii.edu/*, 2006.

Wyrick, G., and R. Hipp, SQLite, *http://www.sqlite.org/*, 2000–.

K. Schwehr, Center for Coastal and Ocean Mapping, University of New Hampshire, Chase Ocean Engineering 24 Colovos Rd, Durham, NH 03824, schwehr@ccom.unh.edu, http://schwehr.org