

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:

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

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).

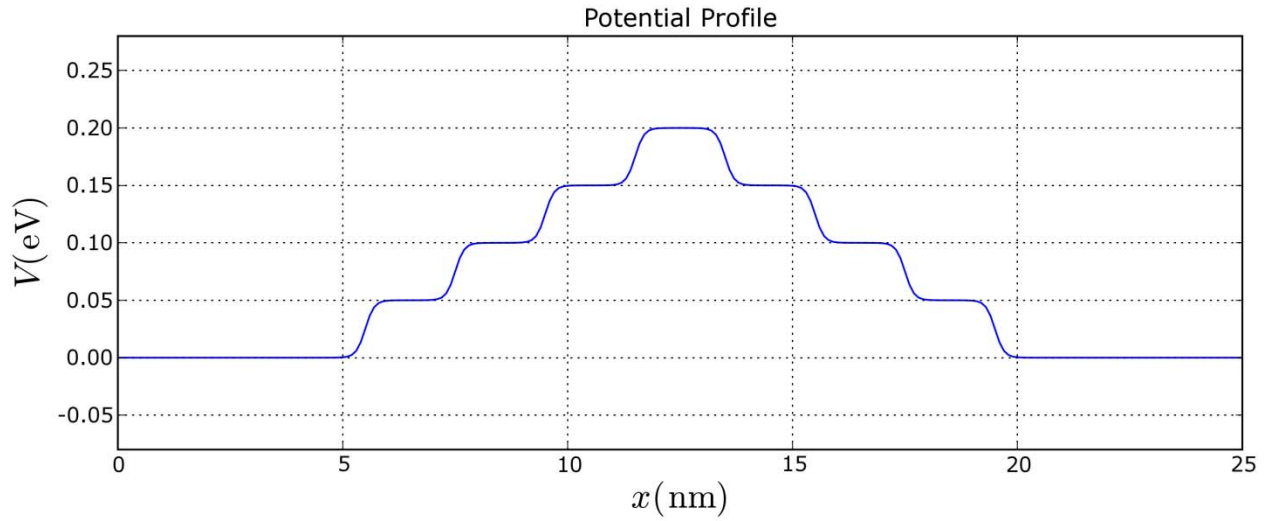


Fig. 1. A typical potential, in units of electronvolt (eV), for a heterostructure lying between 5 and 20 nm ($1 \text{ nm} = 10^{-9} \text{ m}$).

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:

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

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 $k < 0.3 \text{ nm}^{-1}$ and allowing almost complete transmission of electrons with $k > 0.5 \text{ nm}^{-1}$. Further details on the solution of the direct problem can be found in Ref. (1) and the references therein.

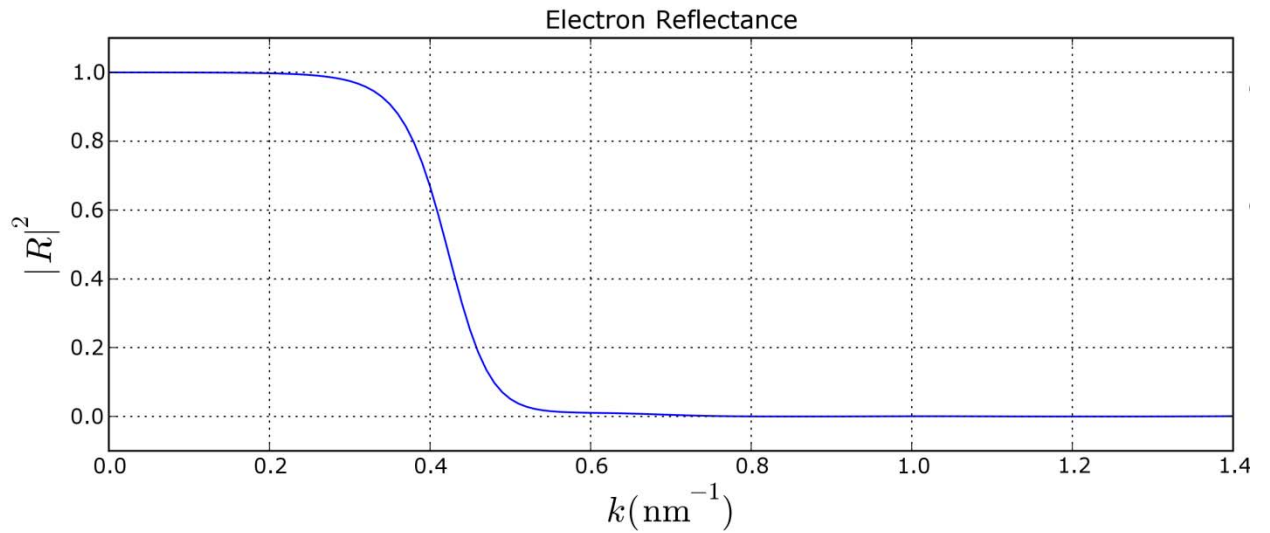


Fig. 2. Calculated electron reflectance as a function of wave vector k , corresponding to the potential in Fig. 1.

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:

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

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:

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

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 $(x, |R(k)|^2, \varphi)$, 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 $(x, |R|^2, \varphi)$ 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 subplots.

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.

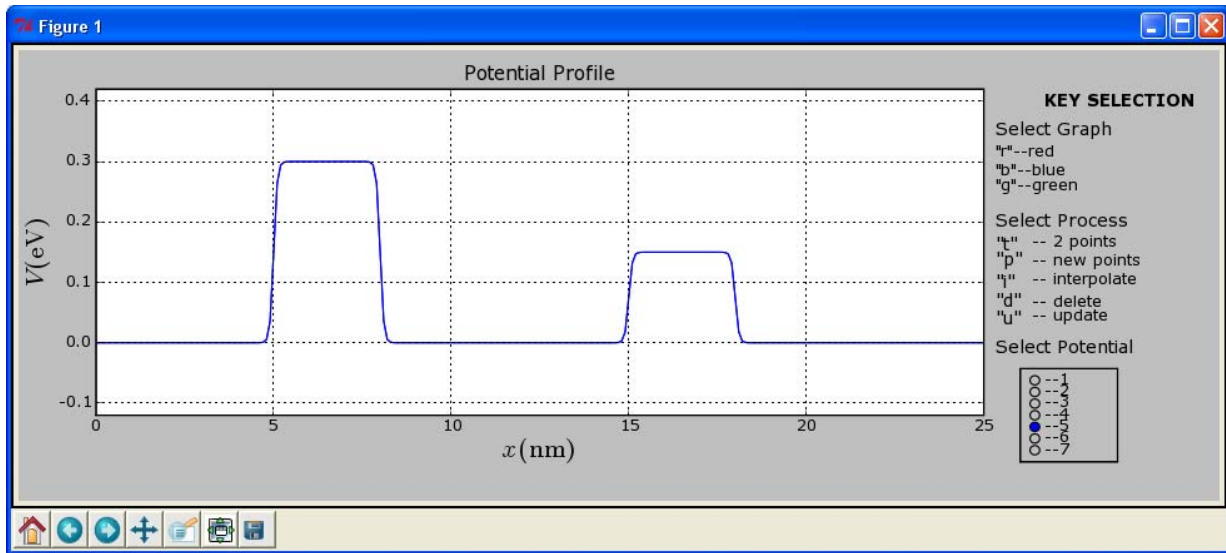


Fig. 3. The display window for the potential. In this simple example the heterostructure consists of three layers which are located between 5 and 17 nm. Notice that the central layer of this heterostructure is made of the same material as the surrounding substrate material.

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.

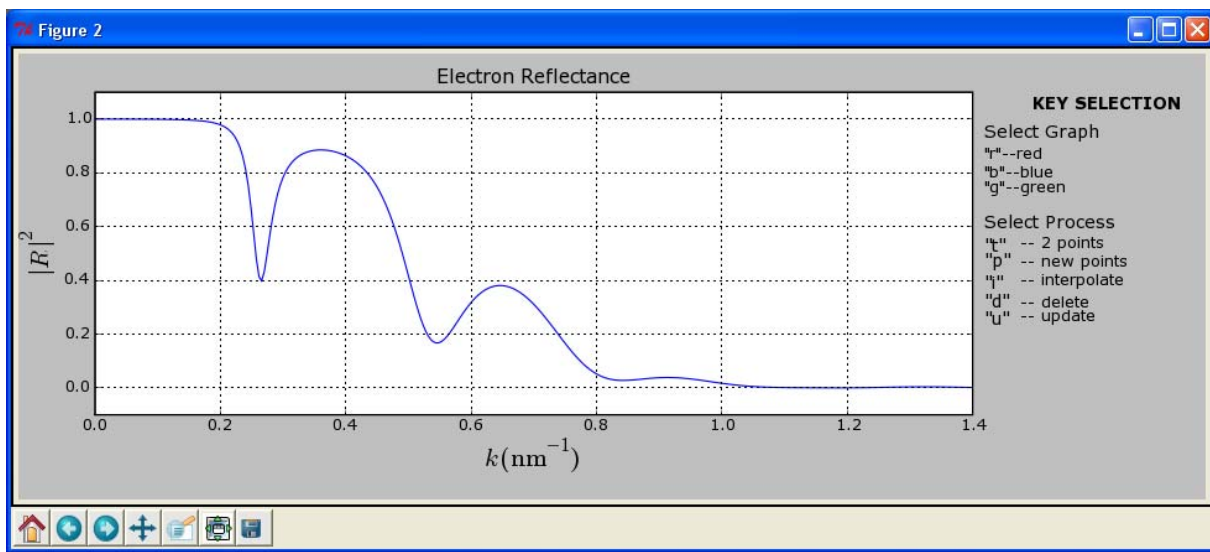


Fig. 4. The display for the electron reflectance corresponding to the potential in Fig. 3.

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:         deltax = 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]))< deltax)):
14:                 temp = ((activeGObject[0].x)[i], ((activeGObject[0].y)[i]))
15:                 flag = True
16:                 break
17:
18:         if ((flag==True) & (countclicks <2)):
19:             thetwopoints.append(temp)
20:             if (activeGObject[0].type == 'profile'):
21:                 offsets1.append(((thetwopoints[countclicks])[0],
22:                                 (thetwopoints[countclicks])[1]))
23:                 figure1.canvas.draw()
24:             elif(activeGObject[0].type == 'reflectance'):
25:                 offsets2.append(((thetwopoints[countclicks])[0],
26:                                 (thetwopoints[countclicks])[1]))
27:                 figure2.canvas.draw()
28:             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.

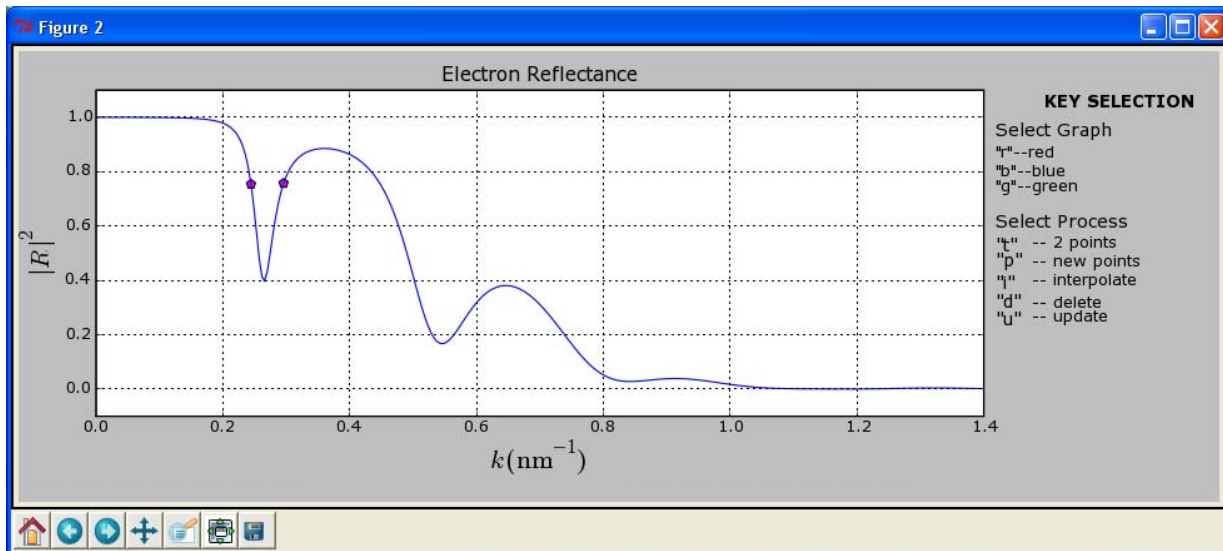


Fig. 5. Electron reflectance, as in Fig. 4, with the two points denoting the interval in which the reflectance is to be modified. In this example the resonance peak between $k = 0.2$ and $k = 0.4 \text{ nm}^{-1}$ will be enhanced to allow almost all electrons with $k = 0.26 \text{ nm}^{-1}$ to pass through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

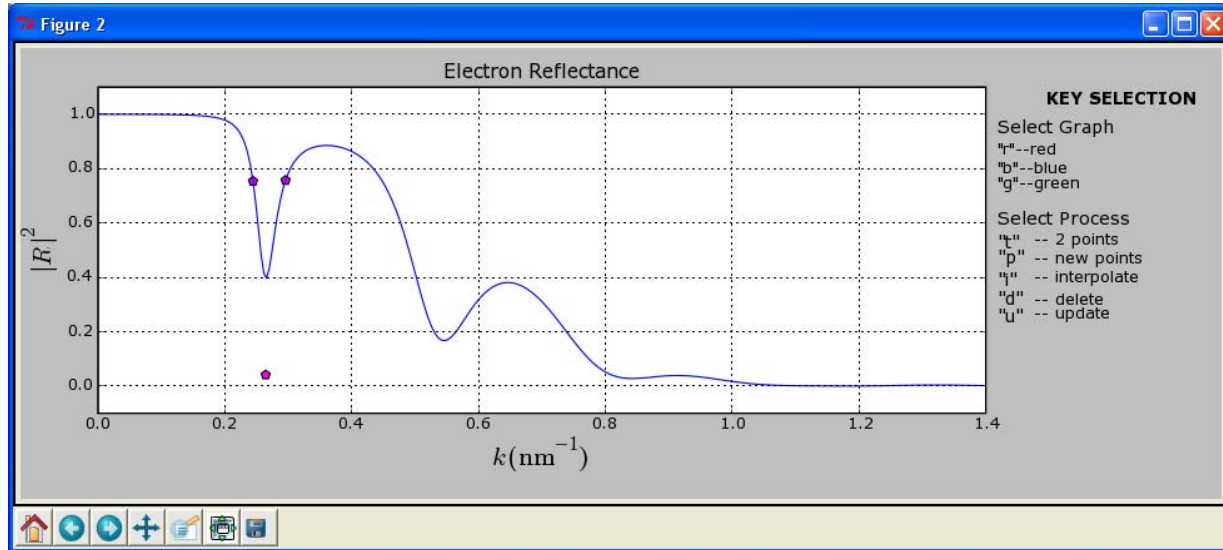


Fig. 6. Electron reflectance, as in Figures. 4 and 5. Here a single point has been chosen between the previously selected two points to enhance the resonance peak at $k = 0.26 \text{ nm}^{-1}$.

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.

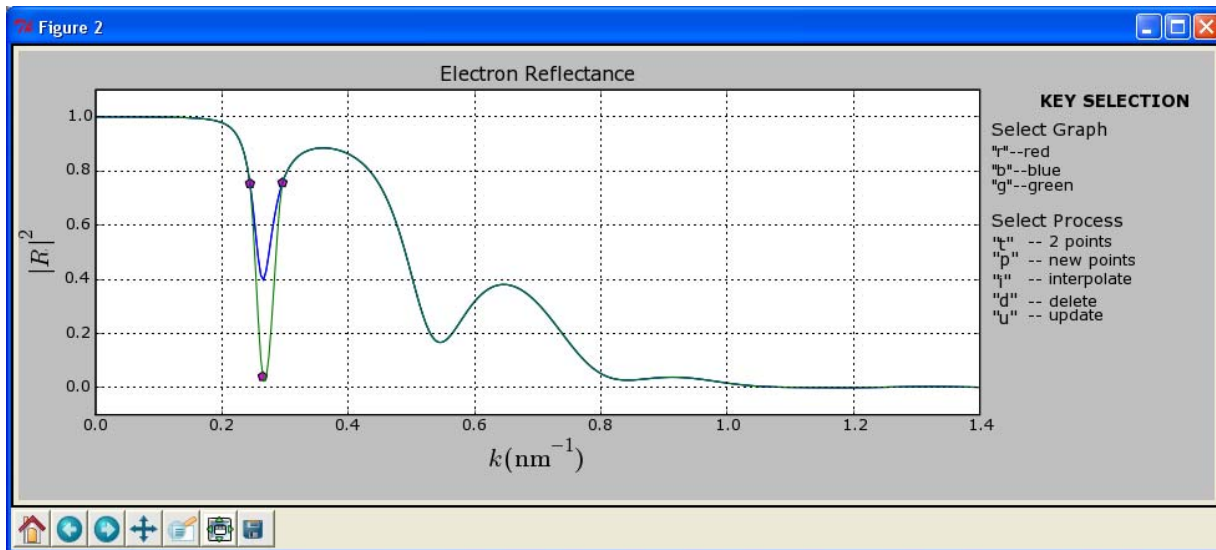


Fig. 7. Electron reflectance, as in Figures 4, 5 and 6. The green curve shows the enhanced resonance peak which has been interpolated through the selected points.

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.

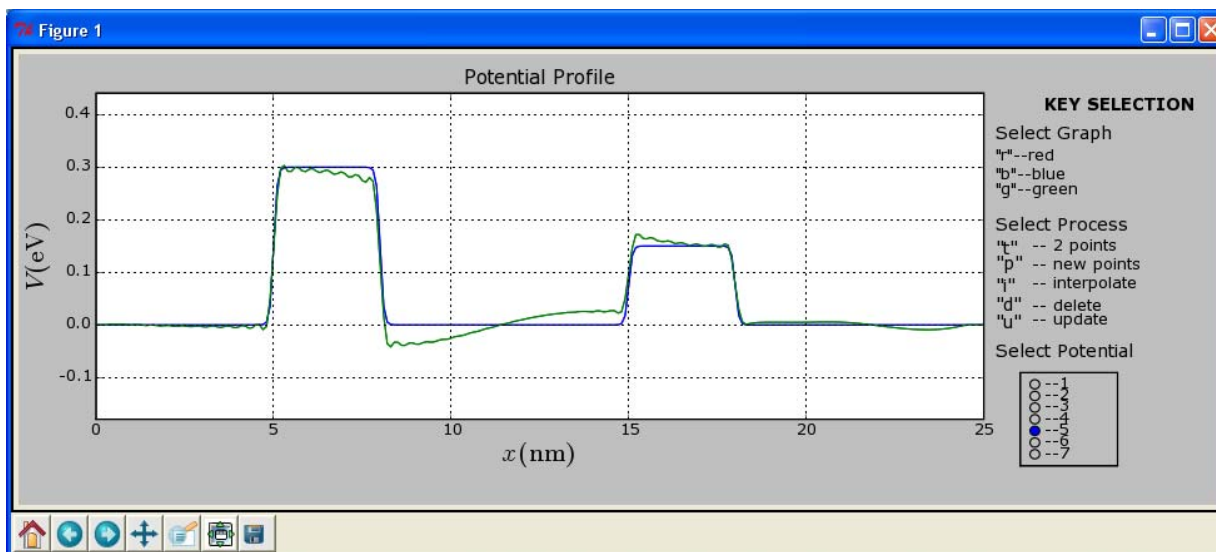


Fig. 8. The original potential (blue) as well as the modified potential (green) which was calculated from the modified reflectance shown in Fig. 7. This result indicates how the potential should be modified (by, for example, modulation doping) in order to achieve the enhanced resonance peak shown in 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 *k-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 : Springer-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.