MODELING AND VISUALIZATION OF SOME PHYSICS PHENOMENA WITH VPYTHON SIMULATIONS†

Godfrey Akpojotor1, Louis Ehwerhemuepha2, Myron Echenim1 and Famous Akpojotor3

1Physics Department, Delta State University, Abraka, Nigeria
2Physics Department, University of Lagos, Lagos, Nigeria
3Physics Department, University of Benin, Benin City, Nigeria

Corresponding author: akpogea@delsung.net1

Abstract

Python unique features such as its interpretative, multiplatform and object oriented nature as well as being a free and open source software creates the possibility that any user connected to the internet can download the entire package into any platform, install it and immediately begin to use it. Thus Python is gaining reputation as a preferred environment for introducing students and new beginners to programming. Therefore in Africa, the Python African Tour programme has been launched and one of us (GEA) is coordinating its use in computational science. One of the current projects in computational programme is using Python as a programming tool for modeling, simulation and visualization of Physics phenomena. In this paper, therefore, we present an overview of the use of VPython which is the visual module of Python, to model and simulate some simple phenomena in physics with explicit visualization of the behaviours of these phenomena under varying conditions.

Keywords: VPython, programming, modeling, simulation, computational physics


94
1. INTRODUCTION

The use of computers in physics, as well as most other branches of science and engineering, has increased many times along with the rapid development of faster and cheaper hardware as well as faster and less complex algorithms. The fusion of faster hardware and smarter algorithms opens up entirely new opportunities for modeling and simulation of the real world [1-5]. Modeling is the use of programming language or mathematics applications to create an abstract model of an object, organization and processes [1]. In general, a model is an attempt to mimic reality in a controlled environment. In physics, a model can be conceived as a simple and well understand phenomenon or system designed to represent and explain a complex and not well understand phenomenon or system [5]. Therefore the model should possess the domineering traits or properties of the phenomenon or system it is supposed to represent. In other words, one needs to know some of these properties of the complex system before one can design a model for it. The implication is that in computer modelling of a system, we need to know some key parameters that control the system we intend to model. Thus in the phenomena we will consider here, we will explain their physics and the key parameters that the algorithms to be designed to model them will depend on.

In general, simulation is the mapping of a physical system into some alternative physical system, which can be easily manipulated and controlled, yet it is described by the same mathematics. One example is the recently developed and growing branch of physics wherein cold atoms in optical lattices is used to simulate phenomena in condensed matter physics [6-7] and quantum computers [8]. The more common simulation is the use of computer to implement a model in order to test it under different conditions with the objective of learning about the model’s behaviour [2]. Thus the results from the computer simulation will determine how well the model represents the system it is designed to represent. The implication is that the simulation process needs to be effective so that it will give a true behaviour of the model. Since this simulation has to do with writing a programme, the choice of programming language becomes very important.

Our choice of programming language is Python which is gaining the reputation as the popular choice of a programming environment for a first exposure to computation in science and engineering [4-5, 9-12]. The reason is that Python has a clean design and object-oriented nature. Furthermore, it has a comprehensive library of modules not only for scientific computation but also for tasks as diverse as manipulating images and running web server as well as the software for mobile devices such as the Nokia series, iphone and ipod. Currently, much of the programming done by Google is in Python (for a list of some other applications of Python, see p8 of Ref. [13]). Even more compelling, Python is open source, multi-platform and free. Therefore, any user connected to the internet can
Two of the important modules for simulation and visualization with Python are VPython and Matplotlib. The VPython which is a combination of Python plus a module called Visual is a users’ friendly environment for easy creation of navigable 3D simulations and also has a captivating ability to manipulate 3D vectors mathematically. Thus the VPython is a simple and an accessible but yet powerful computational tool for creating effective simulation with explicit visualization. The Matplotlib is the Python module for plotting, data analysis and data visualization. It emulates the plotting capabilities of MATLAB which is proven to be an adequate plotting tool. We will use the VPython to model, simulate and visualize the following phenomena: (1) simple pendulum so as to aid the teaching of the simple pendulum experiment or play even an alternative role to the experiment in those parts of the world where the facilities for doing experimental physics are often inadequate at best and not even available in other cases and (2) Fourier transform so as mimic this process which is very important for scientists and engineers studying electrical signals, frequencies, etc. The Matplotlib will be used to model, simulate and visualize quantum wavepackets so as to provide visualization of this often exoteric aspect of physics.

2. SIMULATING THE SIMPLE PENDULUM EXPERIMENT

The simple harmonic motion (SHM) is that in which a point mass that is displaced from its equilibrium position, undergoes successive vibrations before returning to its initial state owing to the force attracting it to the equilibrium position. Also known as simple harmonic oscillation (SHO), it is one of the most fundamental problems in physics. Though all its solutions are yet unknown, but many of them are already known. Since the goal of physics is to explain all physical phenomena irrespective of their complexities, with the few known ones, the SHM enables us to study complicated systems in which the interparticle forces are linear functions relative to the displacement of particles from their equilibrium. The common example of a SHM (see Fig. 1) is the motion of a mass on a spring when it is subject to the linear elastic restoring force given by Hooke's Law: this law states that [14]

\[ F = -kx \]  

(1)

where \( k \) is the force constant.
The resultant motion is sinusoidal in time and demonstrates a single resonant frequency. It is expressed mathematically as

\[ y = A \sin \omega t \]

\[ = A \sin \sqrt{\frac{k}{m}} t \]

(2)

where \( A \) is the amplitude of the sinusoidal motion, \( t \) is the time and \( \omega \) is its angular frequency which can also be expressed in terms of the mass of the spring \( m \) and the force constant \( k \). It is related to the linear frequency by

\[ f = \frac{\omega}{2\pi} \]

(3)

which is in turn related to the period of the sinusoidal motion by \( T = \frac{1}{f} \). The period time is the time taken for one complete oscillation.

One of the simplest ways to experimentally demonstrate the simple harmonic motion is by using the simple pendulum which can be considered to be a point mass suspended from a string or rod of negligible mass (see Fig. 2). Therefore the two parameters that govern the period \( T \) of the simple harmonic motion of the simple pendulum are the length of the string or rod \( L \) and the angle of displacement \( \theta \), that is,

\[ T = 2\pi \sqrt{\frac{L}{g}} \].

(4)
For a displacement $\theta$ from its equilibrium position at the centre, the restoring force that brings it back to the equilibrium is

$$F_{net} = -mg \sin \theta.$$  \hfill (5)

If the displacement is small, then $\sin \theta \approx \theta$ so that

$$F_{net} = -mg \theta.$$  \hfill (6)

The differential form of this equation is

$$\frac{d^2\theta}{dt^2} + \frac{g}{L} \theta = 0$$  \hfill (7)

whose solution is

$$\theta = \theta_{\text{max}} \sin \sqrt{\frac{g}{L}} t.$$  \hfill (8)

which compares with Eq. (2) with $\omega = \sqrt{\frac{g}{L}}$.

A programme was written for the simple pendulum as shown below. The simple pendulum programme has been written to exhibit the elegant feature of Python which is the readability, understanding, maintenance and updating of the python
codes. Every activity is introduced first with the pound sign (#) which denotes the beginning of a comment so that all characters between # and the end of the line are ignored by the interpreter. For example, the creation of the scene is introduced with # Controlling the dimension of the screen. This scene which has its background colour to be white and the scene title as 'Simple Pendulum Visualization' has both the visualization of the swinging pendulum and a graph of the emergent sinusoidal motion. See the snapshots for the length L = 10 and L = 2 in Fig. 3 and the periodic time for L = 10, 8, 6, 4, 2 in Table 1.

```python
from __future__ import division
from visual import *

# Controlling the dimension of the screen
scene.width = 1000
scene.height = 1000
scene.background = color.white
scene.title = 'Simple Pendulum Visualization'

# Turning autoscale off
scene.autoscale = False

# Length of string
length = 2.0

# Setting up the ball and string in an appropriate position
ball = sphere(radius=0.5, pos=(length,0,0), color=color.red)
string = cylinder(pos=(0,0,0), axis=(length,0,0), radius=0.03)
string.color = color.black
ceiling = box(pos=(0,0.5,0), size=(2*length, 1, 0))

# Setting gravity, angular frequency, time and the time step-size
```
gravity = 9.8
dt = 0.01
omega0 = 0
theta0 = 0.5*math.pi

label(pos=(-8,-7,0), text='The length of the string is: %1.5f % length)
label(pos=(-8,-6,0), text='Radius of ball is: %1.2f % ball.radius)

# Creating a list to hold the values of omega, theta and time
omega = [omega0]
theta = [theta0]
time = [0]

# Setting Up the Corresponding Graph Region
graph1 = curve(color = color.green)
x_axis = curve(x=[-12,10], y = [5,5], color=color.black)
y_axis = curve(x=[-10,-10], y = [10,0], color = color.black)
x_label = label(text='Time', pos = (10,4.5,0), opacity=0, color=color.black)
x_label.box = False
y_label = label(text='Theta', pos=(-10.5,8,0), opacity=0, color=color.black)
y_label.box = False

# Setting the "While" Loop Invariant
Loop = True

# Counter Variable to Keep Track of the Number of Loops Executed
i = 0

# Implementing a simple pendulum algorithm
while Loop:
    rate(100)

    # Determining Omega and Theta Values
    omega.append(omega[i] - (gravity/length) * theta[i]*dt)
    theta_modified = theta[i] + omega[i+1] * dt

    # Reseting Theta to Keep It Within a 180 Degrees Range
    if theta_modified > pi:
        theta_modified = theta_modified - pi
    elif theta_modified < -pi:
        theta_modified = theta_modified + pi

    theta.append(theta_modified)
    time.append(time[i] + dt)

    # Updating the Position of the Ball and String
    x = length * math.sin(theta[i])
    y = -length * math.cos(theta[i])
    ball.pos = (x,y,0)
    string.axis = (x,y,0)

    # Updating the Graph
    graph1.append(pos=(time[i]-10,2*theta[i]+5,0))

    # Print the Times the Graph Cuts the X-axis
    if theta[i] > -0.01 and theta[i] < 0.01:
        time_indicator = label(text='%.2f' % time[i], pos=(time[i]-10,4.5,0))
        time_indicator.color = color.black
time_indicator.opacity = 0
time_indicator.box = False

# Updating the Counting Keeping Track of the Number of Loops Executed
i += 1

# Determining When to End the Simulation
if i >= 1850:
    Loop = False
    end_label = label(text='End of Simulation', pos=(0,-8,0))
    ball.pos = (0,-6,0)
    string.axis = (0,-6, 0)

Fig. (3a) (colour online) A snapshot of the VPython programme of a simple pendulum for the length of the string $L = 10.00$ at the end of the simulation.
Table 1. The lengths L and corresponding period time T recorded from the simulated simple pendulum programme.

<table>
<thead>
<tr>
<th>Length (L)</th>
<th>Period (T)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.00</td>
<td>3.17</td>
</tr>
<tr>
<td>8.00</td>
<td>2.84</td>
</tr>
<tr>
<td>6.00</td>
<td>2.46</td>
</tr>
<tr>
<td>4.00</td>
<td>2.00</td>
</tr>
<tr>
<td>2.00</td>
<td>1.42</td>
</tr>
</tbody>
</table>
3. SIMULATING THE QUANTUM MECHANIC WAVEPACKETS

Quantum physics can be regarded as the fundamental theory of microscopic (that is, systems with sizes that are $\leq atom = 10^{-10}m$) phenomena. Prior to 1900, many of the phenomena and experiments in physics can conveniently be accounted for with Newtonian physics also now known as classical mechanics. Such phenomena are classified today as macroscopic (that is, systems with sizes that are $> atom = 10^{-10}m$). These include the motion of mechanical objects both at terrestrial and celestial scales which can be accounted for using the Newton’s equations of motion. Also, the application of the Newtonian mechanics to molecular motions gave birth to the kinetic theory of gases. The inadequacy of classical physics becomes obvious in two quests: to account for phenomena such as the blackbody radiation spectrum, specific heat of solids at low temperature, photoelectric effect, Compton effect, atomic spectra and later the x-rays and radioactivity. The second was the quest to develop an atomic model which could account for the atomic spectra. Thus quantum physics was developed from the experimental data of phenomena that are almost entirely beyond the range of direct human perception [15]. Consequently, quantum physics encompasses physical concepts and interpretations that are alien to our every day experience, so much so that it often seems not to appeal to common sense. Consequently it is difficult to construct mental models to visualize most of the phenomena we study in quantum mechanics [16].

The Schrödinger’s equation (SE) is one of the most important equations in quantum mechanics today as it gives the dynamics of the physical system. For example, it is solved to determine the wavefunction which contains all the information that can be known about the system. The derivation of the SE is a product of scientific boldness to extend the frontiers of knowledge. First, Schrödinger expressed the Broglie’s hypothesis concerning the wave behaviour of matter,

$$\lambda = \frac{h}{p}, \quad (9)$$

in a mathematical form that is adaptable to a variety of physical problems without additional arbitrary assumptions, that is,

$$p = hk \text{ by using } k = \frac{2\pi}{\lambda} \text{ and } E = h\omega \quad (10)$$

by using $E = hr$ and $\omega = 2\pi r$. 

104
Next, Schrödinger set out to find a wave equation for matter that would give particle-like propagation when the wavelength becomes comparatively small just as in the case in optics in which the straight-line propagation of light rays can be derived from wave motion when the wavelength is small compared to the dimensions of the apparatus employed. The final equation that emerged which gave birth to wave mechanics is the time dependent Schrödinger equation:

\[ i\hbar \frac{d}{dt}\psi(r,t) = -\frac{\hbar^2}{2m} \nabla^2 \psi(r,t) + V(r,t)\psi(r,t). \] (11)

In a stationary state, this equation does not depend on time and can therefore be expressed as

\[ E\psi(r,t) = -\frac{\hbar^2}{2m} \nabla^2 \psi(r,t) + V(r,t)\psi(r,t) \] (12)

which is the Schrödinger equation that is time independent.

To study the connection between classical mechanics and quantum mechanics, Schrödinger proposed a wave packet solution for localized particles to his equation [17]. The behavior of the propagation of this wavepacket when it encounters a potential barrier \( V \) can be used to distinguished a quantum particle from the classical particle. In classical mechanics, if the energy of the particle \( E \) is lesser than the maximum height of the potential barrier (i.e. \( E < V \)), the particle will be reflected back at this barrier; if \( E > V \), the particle escapes. In quantum mechanics, however, the situation is not so simple. The particle can be transmitted across the barrier even if its energy \( E \) is below the height of the barrier \( V \), although the probability of escape is small unless \( E \) is close to \( V \). In that case, the particle may tunnel through the potential barrier and emerge with the same energy \( E \). Thus one early counterintuitive observation is that the wavepacket of a quantum particle can penetrate into a classically forbidden region and appear beyond a potential barrier that it does not have enough energy to overcome.

Creating a mental picture of the quantum particle encountering a barrier is often difficult. Therefore simulating and visualizing it with a MatPlotlib programme so that students can visualize the propagation of the wavepacket encountering a step potential for various values of \( E \) and \( V \) is quite captivating. In the programme below, a constant wavepacket energy \( E = 0.0124932555014 \) and a variable Potential height \( V \) are the key features of the simulation. As depicted in the snapshots in Fig 4a, the wavepacket is completely reflected at the step potential since the \( V = 0.05 \) is greater than the \( E \) while in Fig. 4b, the wavepacket is transmitted since the potential \( V = 0.009 \) is less than the \( E \). Finally, Fig. 4c is the snapshots of the wavepacket that is partly reflected and partly transmitted when
the potential $V = 0.013$ is slightly greater than $E$. For the programming details, see Ref. [5].

# Numerical and plotting libraries
import numpy as np
import pylab

# Set pylab to interactive mode so plots update
pylab.ion()

# Gaussian pulse function to act as an envelope to the wave function.
def Gaussian(x, t, sigma):
    """ A Gaussian curve.
    $x = \text{Variable}$
    $t = \text{time shift}$
    $\sigma = \text{standard deviation}$
    """
    return np.exp(-(x-t)**2 / (2 * sigma ** 2))

def free(npts):
    """Free particle."
    return np.zeros(npts)

def step(npts, POTENTIAL_HEIGHT):
    """Potential step"
    v = free(npts)
    v[npts / 2:] = POTENTIAL_HEIGHT
    return v

def fillax(x, y, * args, ** kw):
    """Fill the space between an array of $y$ values and the $x$ axis.
    All args/kwargs are passed to the pylab.fill function.
    Returns the value of the pylab.fill() call.
    """
    xx = np.concatenate((x, np.array([x[-1], x[0]], x.dtype)))
    yy = np.concatenate((y, np.zeros(2, y.dtype)))
    return pylab.fill(xx, yy, * args, ** kw)

N = 1600  # Number of spatial points.
T = 5*N   # Number of time steps.
\[ T_p = 50 \quad \# \quad \text{Number of time steps to increment before updating the plot.} \]
\[ dx = 1.0e0 \quad \# \quad \text{Spatial resolution} \]
\[ m = 1.0e0 \quad \# \quad \text{Particle mass} \]
\[ \text{PlackConstant} = 1.0e0 \quad \# \quad \text{Plank's constant} \]
\[ X = dx \times \text{np.linspace}(0, N, N) \quad \# \quad \text{Spatial axis.} \]
\[ \text{POTENTIAL\_HEIGHT} = 1.0e-2 \quad \# \quad \text{Potential amplitude (used for steps and barriers)} \]
\[ \text{BARRIER\_THINKNESS} = 5.0e-2 \quad \# \quad \text{"Thickness" of the potential barrier. By reducing the barrier thinkness reflection is seen.} \]
\[ \text{POTENTIAL} = \text{'}step\text{'} \]
\[ \text{sigma} = 40.0 \quad \# \quad \text{Standard deviation on the Gaussian envelope.} \]
\[ \text{TIME\_SHIFT} = \text{round}(N / 2) - 5 \times \text{sigma} \quad \# \quad \text{Time shift} \]
\[ \text{WAVENUMBER} = \pi / 20 \quad \# \quad \text{Wavenumber (note that energy is a function of k)} \]
\[ E = \left( \text{PlackConstant} \times 2 / 2.0 / m \right) \times \left( \text{WAVENUMBER} \times 2 + 0.5 / \text{sigma} \times 2 \right) \]
\[ \text{if POTENTIAL} == \text{'}step\text{'}:\]
\[ V = \text{step}(N, \text{POTENTIAL\_HEIGHT}) \]
\[ \text{else:} \]
\[ \text{raise ValueError(\text{"POTENTIAL TYPE UNKNOWN: \%s\" \% POTENTIAL\})} \]
\[ \# \quad \text{Additional parameters} \]
\[ \text{POTENTIAL\_max} = V.\text{max()} \quad \# \quad \text{Maximum potential of the domain.} \]
\[ \text{dt} = \text{PlackConstant} / \left( 2 \times \text{PlackConstant} \times 2 / \left( m \times dx \times 2 \right) + \text{POTENTIAL\_max} \right) \# \quad \text{Critical time step.} \]
\[ c1 = \text{PlackConstant} \times \text{dt} / \left( m \times dx \times 2 \right) \]
\[ c2 = 2 \times \text{dt} / \text{PlackConstant} \]
\[ c2V = c2 \times V \]
\[ \text{print \text{’ONE DIMENSION SCHRODINGER WAVE EQUATION FOR A STEP POTENTIAL’} \]
\[ \text{print \text{’WAVEPACKET ENERGY: \’, E} \]
\[ \text{print \text{’POTENTIAL TYPE: \’, POTENTIAL} \]
\[ \text{print \text{’POTENTIAL\_HEIGHT: \’, POTENTIAL\_HEIGHT} \]
\[ \text{print \text{’BARRIER\_THINKNESS: \’, BARRIER\_THINKNESS} \]
\[ \# \quad \text{Wave functions. Three states represent past, present, and future.} \]
\[ \text{psi\_r} = \text{np.zeros}(3, N) \# \quad \text{Real} \]
\[ \text{psi\_i} = \text{np.zeros}(3, N) \# \quad \text{Imaginary} \]
 psi_p = np.zeros(N, )   # Observable probability (magnitude-squared of the complex wave function).

# Time constants used for accessing rows of the wavefunctions.
PA = 0   # Past
PR = 1   # Present
FU = 2   # Future

xn = range(1, N / 2)

x = X[xn] / dx   # Normalized position coordinate

gauss = Gaussian(x, TIME_SHIFT, sigma)

cx = np.cos(WAVENUMBER * x)
sx = np.sin(WAVENUMBER * x)

psi_r[PR, xn] = cx * gauss
psi_i[PR, xn] = sx * gauss
psi_r[PA, xn] = cx * gauss
psi_i[PA, xn] = sx * gauss

# Initial normalization of wavefunctions Compute the observable probability.
psi_p = psi_r[PR] ** 2 + psi_i[PR] ** 2

# Normalize the wave functions so that the total probability in the simulation is equal to 1.
P = dx * psi_p.sum()   # Total probability.

nrm = np.sqrt(P)
psi_r /= nrm
psi_i /= nrm
psi_p /= P

# Initialize the figure and axes.
pylab.figure()
xmin = X.min()
xmax = X.max()

ymax = 2.3 * (psi_r[PR]).max()

pylab.axis([xmin, xmax, -ymax, ymax])

lineR, = pylab.plot(X, psi_r[PR], 'b', alpha=0.7, label='Real')

lineI, = pylab.plot(X, psi_i[PR], 'r', alpha=0.7, label='Imag')

lineP, = pylab.plot(X, 6 * psi_p, 'k', label='Prob')
pylab.title('Potential height: %.2e' % POTENTIAL_HEIGHT)
# energy, in the same units the potential is being plotted.
if POTENTIALmax != 0:
    # Scaling factor for energies, so they fit in the same plot as the
    # wavefunctions
    Efac = ymax / 1.0 / POTENTIALmax
    V_plot = V * Efac
    pylab.plot(X, V_plot, ':k', zorder=0)  # Potential line.
    fillax(X, V_plot, facecolor='y', alpha=0.2, zorder=0)
    pylab.axhline(E * Efac, color='g', label='Energy', zorder=1)
    pylab.legend(loc='lower right')
    pylab.draw()

pylab.xlim(xmin, xmax)
IDX1 = range(1, N-1)             # psi [ k ]
IDX2 = range(2, N)              # psi [ k + 1 ]
IDX3 = range(0, N-2)            # psi [ k - 1 ]
for t in range(T + 1):
    psi_rPR = psi_r[PR]
    psi_iPR = psi_i[PR]
    # Apply the update equations.
    psi_i[FU, IDX1] = psi_i[PA, IDX1] +
        c1 * (psi_rPR[IDX2] - 2 * psi_rPR[IDX1] +
              psi_rPR[IDX3])
    psi_i[FU] -= c2V * psi_r[PR]
    psi_r[FU, IDX1] = psi_r[PA, IDX1] -
        c1 * (psi_iPR[IDX2] - 2 * psi_iPR[IDX1] +
              psi_iPR[IDX3])
    psi_r[FU] += c2V * psi_i[PR]
    # Increment the time steps.  PR -> PA and FU -> PR
    psi_r[PA] = psi_rPR
    psi_r[PR] = psi_r[FU]
    psi_i[PA] = psi_iPR
psi_i[PR] = psi_i[FU]

# Only plot after a few iterations to make the simulation run faster.
if t % Tp == 0:
    # Compute observable probability for the plot.
    psi_p = psi_r[PR] ** 2 + psi_i[PR] ** 2
    # Update the plots.
    lineR.set_ydata(psi_r[PR])
    lineI.set_ydata(psi_i[PR])
    lineP.set_ydata(6 * psi_p)
    pylab.draw()

# So the windows don’t auto-close at the end
pylab.ioff()
pylab.show()

Fig 4a. (colour online) Snapshot of the wavepacket showing that it is completely reflected at the step potential when the potential V = 0.05 is greater than the wavepacket energy E = 0.0124932555014.
Fig 4b. Snapshot showing that the wavepacket is completely transmitted at the step potential $V = 0.009$ slightly less than the wavepacket energy $E = 0.0124932555014$. 
4. SIMULATING THE FOURIER TRANSFORM

The Fourier transform is a mathematical operation that decomposes a signal into its constituent frequencies. It is therefore a generalization of the complex Fourier series [18]. A Fourier series is an expansion of a periodic function $f(x)$ in terms of an infinite sum of sines and cosines [19]. It follows that one basic goal of the Fourier transform is to convert complex wave forms into a sinusoidal form (i.e. the sin or cosine waveforms). Its application is in wave motions such in frequency analysis and many other applications in physics and engineering.

Every simulation and visualization has a purpose which determine the modeling. Since the purpose of this last Python programme is to mimic the Fourier transform as schematically shown in Fig. 5, the game here is to create the input waveform (various types) from their equations and then transform them into a sinusoidal waveform. Therefore, we have hidden the actual transformation in a dark box so that the visualization is simply that any input wave form is transformed into a sinusoidal wave as depicted in Fig. 6. Thus the aspect of the programme for the actual transformation is replaced by simply creating a dark box.

![Fig 4c. Snapshot showing that the wavepacket is partly reflected and partly transmitted at the step potential $V = 0.013$ is slightly greater than the wavepacket energy $E = 0.0124932555014$](image)
By using their equations

```python
from visual import *

# Setting Up the Visualization Scene
scene.title = "Mimicking Fourier Transform"
scene.width=600
scene.height=600
scene.background = color.white

# Code for Developing a Step-like Function
step_fn_data = [1]
for i in arange(1,10):
    if math.fmod(i,2) == 0:
        switching_value = [-1,1,1]
    else:
        switching_value = [1,-1,-1]
```

Fig. 5. A schematic diagramme of the Fourier transform wherein the actual transform process is hidden in a dark box.
step_fn_data.extend(switching_value)
d = curve(x = arange(-40, size(step_fn_data)-40), y = step_fn_data)
d.color = color.green

# End - Code for Developing a Step-like Function

# Code for Developing a Red "Blackbox"
black_box = box(pos=(0,0,0), size=(4,20,0), color=color.red)

# Defining Variables to be Used in Developing and Updating the Resultant Waveform
hx_incre = 0
h = curve(color=color.blue)

# Preventing Automatic Scaling of the Visualization Window
scene.autoscale = False

# Adding Informative Label on the Visualization Window
input_wave_info = label(text='The input wave is the green wave', pos=(0,28,0))
input_wave_info.box = False
input_wave_info.color = color.black
input_wave_info.opacity = 0
output_wave_info = label(text='The output wave is the blue wave', pos=(0,23,0))
output_wave_info.box = False
output_wave_info.color = color.black
output_wave_info.opacity = 0
blackbox_info = label(text='The red barrier is the Fourier Mimicry "Blackbox"', pos=(0,18,0))
blackbox_info.box = False
blackbox_info.color = color.black
blackbox_info.opacity = 0

# Setting an Invariant to be Used for the "while" Loop
looping = True
# Controlling the Simulation in a "while" Loop

while looping:
  rate(300)
  if d.x[d.x.argmax()] >= 0:
    d.x[d.x.argmax()] = 0
    d.x +=0.01
  if d.x[d.x.argmax()] >= -2:
    hx_incre+=0.01
    h.append(pos=(hx_incre,2*sin(hx_incre),0))
  if d.x[d.x.argmin()] == 0.01:
    looping = False
    end_of_simulation_info = label(text='End of Simulation', pos=(0,-18,0))

---

**Fig. 6** (colour online) A snapshot showing the execution of the VPython programme mimicking Fourier transform with a green input wave (see insert (b)) and the blue output wave (see insert (c)).
5. SUMMARY AND CONCLUSION

As stated in the preface of Ref. [5], the 21st century researchers in science and engineering cannot fully function if they do not possess some computational knowledge to work with both in theory and experiment. The reason is that computational approaches provide a testing ground for some analytical models [20], a useful tool for the interpretation of experimental results and a powerful technique to predict properties of physical systems that are not easily accessible to experiments [21]. It is hoped that the insight we have given here on how powerful Python can be used as a programming environment will inspire its adoption in computational approaches in science and engineering not only for new beginners but also for those already into programming but need a programming language that is more elegant and all purpose.

Acknowledgement

We acknowledge partial funding by Kusmus Communications, ICBR and AFAHOSITECH. One of us (GEA) appreciates the National Mathematical Centre, Abuja for supporting the inclusion of a foundation postgraduate course on Introduction to computational approaches with Python at ISOTPAND2010.

REFERENCES


