Numerical Approximation Module
Student Guide

Concepts of this Module

- Introduction to the *Python* programming language.
- Numerical approximation as an alternative to analytic solutions.

Introducing Python

Here we briefly introduce the *Python* language and some of the programming constructs that will be used in the main part of this Module.

The *Python* programming language is free and open source, with a huge community of developers. Although it is an ideal first language to learn, you may wish to know that it is not a "toy". It is used extensively by Google, NASA, the Large Hadron Collider just being lit up in Switzerland, Youtube, Air Canada, and many more.

Traditionally the first computer program simply prints *hello, world*. Here is a complete *Python* program that does this:

```
print "hello, world"
```

Here is another complete program that also prints *hello, world*:

```
what = "world"
print "hello,", what
```

The first line of this program assigns *world* to a variable named **`what`**. The next line then prints *hello,* followed by whatever the variable named **`what`** is set to, *world* in this case. The *Python* interpreter executes the lines of this "program" in order.

Today we will wish to have *Python* execute some lines of the program over and over again. We will use a **`while`** loop to do this. This loop has the form:

```
while something_is_true:
        execute this line of the program
        then execute this line of the program
        then execute this next line of the program
```

After executing the third line after the **`while`** statement, it goes back to the **while** statement: if *something* is still true then it executes the following lines again, and so on.

We have prepared a program named `LoopDemo.py` which demonstrates this loop. Here is a listing of the program.

Listing of LoopDemo.py

```
# All lines like this one that begin with a "#"
# are comments. All other non-blank lines are
# program statements.

# Set a variable named "x" to a value of 0
x = 0

while x < 3:
    print x
    # Increase the value of x by one.
    x = x + 1
# End of the while loop. Go back to
# the while statement again.
```

You may wish to know that the lines following the **`while`** statement must be indented as shown.

Start the `IDLE for VPython` program. Use `File` / `Open` … to open the file `LoopDemo.py` which is located in `Feynman:Public/Modules/NumerApprox` folder.

Predict what will happen when this program is run.

Check your prediction by running the program: use `Run` / `Run Module` or press the F5 key on your keyboard.

Sometimes we wish to use a **`while`** statement to have the program execute the same lines over and over until it is manually stopped. The `LoopDemo2.py` file in the same directory does exactly this. A listing of this program is in Appendix 1.

Predict what will happen when this program is run.

Check your prediction by running it.

Also in the `Feynman:Public/Modules/NumerApprox` folder is the file `LoopDemo3.py`, and a code listing is in Appendix 2. It differs from `LoopDemo2.py` in two ways:

1. The first **`print t`** statement is removed.
2. Inside the **`while`** loop the two statements that increment the value of the time and prints the value of the time are reversed.

Predict what will happen when this version is run. Check your prediction by opening the file and running it.

The Spring-Mass System and Numerical Approximation

For a mass *m* on a spring with spring constant *k* Newton's Second Law is:

$$\begin{aligned} F &= ma \\ -kx &= m\frac{d^2x}{dt^2} \end{aligned} \tag{1}$$

This is a second-order differential equation, and if one knows enough calculus one can solve it to get:

$$x = ampl\sin(\omega t) \tag{2}$$

where:

$$\omega = \sqrt{\frac{k}{m}}$$

But if one doesn't know enough calculus or just doesn't want to bother with a differential equation, a moderately powerful computer provides a nice alternative. The basic idea is that we will start with the mass at some known position and calculate its acceleration, how fast it is moving and where it will be small *timestep* Δt later, and keep doing this over and over again. Here is how one may do this *numerical approximation*:

1. From the mass' current position *x* we can calculate the acceleration *a* of the mass: $a = -\frac{k}{m}x$
2. If the speed of the mass is *v*, then calculate a new speed $v_{new} = v + a\,\Delta t$.
3. If the position of the mass is *x*, calculate a new position $x_{new} = x + v_{new}\,\Delta t$.
4. Go back to Step 1 and repeat.

Of course, this method is just an approximation. However given a sufficiently powerful computer to do the calculations we can make the approximation as close to correct as we wish by making the timestep Δt sufficiently small.

We have prepared a *Visual Python* (*VPython*[1]) animation which both uses Eqn. 2 and implements the numerical approximation described above.

[1] *VPython* is free, open source, and available for Windoze, Mac, Linux and UNIX from http://www.vpython.org/.

The Activity

A. Open the `IDLE for VPython` program. Use `File / Open` … to open the file `SHM.py` which is located in `Feynman:Public/Modules/NumerApprox`. Use `Run / Run Module` or press the F5 key on your keyboard to start the animation. The upper yellow sphere uses Eqn. 2, and the lower green sphere uses the numerical approximation. Can you see any differences between the motions of the two spheres? For fun you may wish to know that:
   - Holding down the right mouse button and moving the mouse allows you to rotate the view of the animation.
   - Holding down both mouse buttons and moving the mouse up or down allows you to zoom in and out on the animation.

B. For your convenience a listing of the `SHM.py` code is included in Appendix 3 of this document. In the `Feynman:Public/Modules/NumerApprox` folder the file `CodeBig.pdf` also lists the code using big fonts; you may wish to print this file and place the pages on the whiteboard using small magnets. Including empty lines there are 90 lines in the file. How many of them are program statements?

C. Some lines of the code are used only for the animation of the yellow ball; some lines are only for the animation of the green ball; some lines are shared for the animations of both balls; still other lines are commands to control the animation speed, set up the calculation loop, or set the "stage" for the animation. Circle or use a highlighter on all the lines in the code that are used only for the animation of the yellow sphere and label them with **Y**; if a yellow highlighter is available it would be a good choice for this.

D. Preferably using a different color pen or highlighter, circle or highlight all the lines in the code that are used for the animation of both spheres and label them with **B**.

E. Describe in your own words how the program animates the motion of the yellow ball.

F. From the parameter values set in the code calculate the period T of the oscillation. Does your calculated value match the actual period you see in the animations?

G. About 60% down the code listing the maximum amplitude of the motion `ampl` is calculated. Did you circle this in Part C? If not, should you have? Is the calculation correct? (Hint: think about conservation of energy.)

H. Preferably using a third color pen or highlighter circle or highlight all the lines in the code that are used only for the animation of the green sphere and label them with **G;** a green highlighter would be ideal if available. Circle or highlight all the lines that control the animation speed, set up the calculation loop, or set the "stage" for the animation, and label them with **C**; a fourth color pen or highlighter would be nice if possible. Follow the code for all the lines that are used for the animation of the green sphere. Does it surprise you that nowhere in these lines of code does a trig function appear? Explain.

I. In the code for the yellow ball, the value of the time is incremented and then the new position of the ball is calculated. Is this correct? What if those two lines were reversed?

At the end of this Module, you will want to staple your "de-constructed" code into your lab book.

For the Keen

Here are some things you may wish to do. They are not intended to be part of the Activity of this Practical, but instead some things you may wish to explore on your own.

Some systems, particularly chaotic ones, are not analytically solvable: there is no equation that describes the motion. For such systems numerical approximation is the only way that they may be studied. When *VPython* first starts, using the `File` / `Open …` command lists the examples that are shipped with the software. The `doublependulum.py` program in that directory is an example of a chaotic system which is not analytically solvable but here is solved by numerical approximation. The physics behind this animation is fairly formidable, but the basic idea is the same as the `SHM.py` code you used here. There are many other interesting examples that are shipped with the software.

You may also save a copy of the `SHM.py` file and try modifying it by changing some of the parameters set in the code. You will want to know that by default every time you run the program *VPython* first saves the code into the file. Thus you may wish to consider working on a copy of the master file, named perhaps `SHM_work.py`.

One simple change you could make to `SHM.py` involves efficiency. As written determining the yellow sphere's position involves calculating the angular velocity **`sqrt(k/mass)`** for every iteration of the loop. Calculating the value once before entering the loop and then using the calculated value would mean that the program has to perform many less calculations.

Appendix 1 – `LoopDemo2.py` Code Listing

```
# All lines like this one that begin with a "#" are
# comments. All other non-blank lines are program
# statements.

# Import the visual library.
from visual import *

# Set the time
t = 0

# Set the timestep
dt = 1

# Print the current value of the time
print t

# The next line causes the indented lines that follow
# it to be repeatedly executed in the loop. The construct:
#   1==1
# means "is one is equal to one?" which is always true.
# Thus double equal signs like this mean something different
# than a single equal sign, such as is used above to set the
# values of the time and the timestep.
while 1==1:

    # Do one calculation every second
    rate(1)

    # Increment the value of the time and print the result.
    # Here the single equal sign means set the value of t to
    # whatever appears to the right of the equal sign.
    t = t + dt
    print t
# End of the while loop. Go back to the rate(1) statement
# and start over.
```

Appendix 2 – `LoopDemo3.py` Code Listing

```
# All lines like this one that begin with a "#" are comments.
# All other non-blank lines are program statements.

# Import the visual library.
from visual import *

# Set the time
t = 0

# Set the timestep
dt = 1

# The next line causes the indented lines that follow
# it to be repeatedly executed in the loop. The construct:
#   1==1
# means "is one is equal to one?" which is always true.
# Thus double equal signs like this mean something different
# than a single equal sign, such as is used above to set the
# values of the time and the timestep.
while 1==1:

    # Do one calculation every second
    rate(1)

    # Print the time and then increment its value.
    # Here the single equal sign means set the value of t to
    # whatever appears to the right of the equal sign.
    print t
    t = t + dt
# End of the while loop. Go back to the rate(1) statement and
# start over.
```

Appendix 3 – SHM.py Code Listing

```
# All lines like this one that begin with "#" are comments.
# All other lines are program statements.

# The next line is an internal revision control id:
# $Date: 2007/11/08 17:19:19 $ $Revision: 1.2 $
# Copyright (c) 2007 David M. Harrison

# Import the visual library.
from visual import *

# These  four lines control the size of the window of
# the animation and the scale. The details of these lines
# are not important for our purposes.
scene.autoscale = 0
scene.height = 400
scene.width = 800
scene.range = vector(60, 60, 60)

# Create the green ball that will execute simple harmonic motion
# by numerical integration.
greenBall = sphere (color = color.green, radius = 2)

# yellowBall will execute simple harmonic motion using a sine function.
yellowBall = sphere (color = color.yellow, radius = 2)

# The initial x position of the balls: this is
# the equilibrium position.
x = 0

# Position the balls. pos is a built-in of VPython, and
# lists the (x,y,z) coordinates. The x axis is horizontal,
# y axis is vertical, and the z axis is perpendicular to
# the plane of the screen. We place the green ball just
# below the center of the scene, at y - -10.
#
greenBall.pos = (x,-10,0)

# yellowBall is above the first ball: it's y coordinate is 10,
# just above the center of the scene.
yellowBall.pos = (x, 10, 0)

# The initial x component of the velocity of the balls:
# all other components are zero.
vx = 150

# The spring constant
k = 9.0

# The mass of the balls
mass = 1.0

# The amplitude of yellowBall's motion
ampl = sqrt(mass/k) * vx
```

```
# The time
t = 0

# This is the time step
dt = 0.005

# This causes the following indented lines
# to be executed forever in a loop.
while 1 == 1:

    # Set the rate of the animation
    rate(1/dt)

    # The acceleration in the x direction.
    a = -(k/mass) * x

    # Update the speed using the acceleration. Note
    # that we "recycle" the variable vx, replacing the
    # old value with the new one.
    vx = vx + a*dt

    # Update the x position of the ball using the speed.
    x = x + vx*dt

    # Position greenBall at the new x position
    greenBall.pos = (x, -10, 0)

    # Update the time
    t = t + dt

    # Now we calculate simple harmonic motion using
    # a sine function and position yellowBall using the result
    # of the calculation
    x2 = ampl * sin( sqrt(k/mass)* t)
    yellowBall.pos = (x2,10,0)
```

This Guide was written by David M. Harrison, Dept. of Physics, Univ. of Toronto in November 2007.
Last revision: March 4, 2008.