Introduction to Jupyter Notebooks

1 Introduction

In this course, we will be doing some scientific computing, which will allow us to solve problems that we
would not be able to solve with pencil and paper. We will be doing this work using [Jupyter Notebooks]. In
the notebook, we will use Python to to our programs, and in particular, an add on to python known as visual
Python, or [vpython]. You are not expected to have had any programming experience prior to this course.
During most of the course, we will provide program shells that only require you to code the physics, then
run the programs.

2 Installation of Python, Jupyter Notebooks and VPython

It is advisable to install Jupyter Notebooks on your own computer. The easiest way to do this is to use
the Anaconda distribution of Python available at Continuum Analytics. Use the Download link on their
website, [https://www.anaconda.com/](https://www.anaconda.com/) and follow the instructions there for your computer. There is support
Windows, MacOS and Linux.

We will be using VPython, and you will need to install this in addition to anaconda. The instructions
can be found on [vpython.org](http://vpython.org). Once you have installed anaconda, open a Command Prompt (on Windows)
or a terminal window on Mac OS (under Utilities) and Linux. In that window,

```
conda install -c vpython vpython
```

or

```
pip install vpython
```

You can update your installation using the corresponding command to how you installed.

```
conda update -c vpython vpython
```

or

```
pip install vpython --upgrade
```

On a Windows machine, you may also need to install nodejs. To do this, download from [https://nodejs.org/en/download/](https://nodejs.org/en/download/) and then execute the command

```
jupyter labextension install vpython
```

3 Jupyter Notebooks

3.1 Start a Jupyter Notebook

From a Windows machine,

Windows Menu -> Anaconda 3 -> Jupyter Notebook

From a Mac, you can either launch from a terminal window (open in Utilities).

```
jupyter notebook
```

or to specify your browser

```
jupyter notebook --browser=firefox
```

Use can also use Anaconda-Navigator under Applications. On linux, it is the same a the terminal on a Mac.
3.2 Getting Started

The first thing you should do is name your notebook. Click on the Untitled and give your notebook a name. I used “BouncingBall” for this example. Jupyter notebooks are organized into units called “cells”, and is the region to the right of the

In [ ]:

in the figure. The type of cell is indicated by box labeled code. There are several types of cells, but the two that we will use here are

- Markdown cells where we can add and format comments to the notebook.
- Code cells, where we will be able to Vpyton code.

We will start by entering some comments, so we need to change our cell to a Markdown cell. We can now enter text in the markdown cell. and the format it by hitting

Shift+Enter

If you double click on the formatted text, you can edit it.

We can now go to the cell that is just below our comments and enter some code. A simple example is to type
print("Hello World")

into the cell, and then “execute” or “run” the code. You run the code by clicking on the little arrow on the left side of the cell or the Run at the top of the notebook.
3.3 Starting Vpython

We now need to tell our notebook that we want to use vpython. To do this, we need to import a couple of modules by typing the following into our code cell and then running it. We can now start to execute vpython and math expressions. To do this, we will create a new cell below the one we have by clicking on Insert and selecting Insert Cell Below from the dropdown menu. In the new cell you can now use python as a calculator. Type in various things below and see what you get.

```python
> print (12+56)
> print (3**3)
> print (sqrt(25))
> a = vector(1,1,1)
b = vector(-1,2,3)
print (a+b)
pdint (dot(a,b))
```
print (cross(a,b))
print (mag(a))
print (norm(a+b))
> print (sin(2))
> print (cos(pi))

You can also do loops and conditionals. A loop will continue to execute the same piece of code until some condition is met, while a conditional will execute a piece of code if the condition is true. We have a few examples these here. A example of a loop is shown below, loops are started with the while command, and the indentation and punctuation are critical. The loop will continue as long as $i$ is smaller than 5, then exit. The code

```python
i += 1
```

adds one to the value of $i$. It could have also been written as

```python
i = i + 1
```

where both are equivalent. Running this code produces the result as shown. Another type of loop is a so-called infinite loop, that will never stop. An example is given by

```python
> i = 1
while True:
    print (i)
    i+=1
```

Conditionals are started with the if command. A simple example is something like following which will loop over values of $i$ between 1 and 9, and print when $i$ is equal to 4.

```python
> i=1
while i<10:
    if i==4:
        print("i is equal to",i)
    i+=1
```

There are various logical expressions that we can use in both loops and conditional statements. In addition to these comparative test, we can also use and and or in our comparisons.

### 4 Animating Physics
We are now going to move forward to
4.1 3D Geometric Objects

There are a number of three-dimensional objects that vpython know about. A full list can be found at https://vpython.org/contents/docs/primitives.html. We will introduce these shapes as needed throughout the course. The first that we will use is a sphere. If we start by entering

sphere()

and then running the code will produce a new cell above the code cell. This cell contains a grey sphere on a black background as shown below. You can manipulate the object. On a Mac, using two fingers on your track pad zooms in or out on the object. Using Shift and one finger allows you to move the point of view, and Control and one finger allows you to rotate about the center of the screen. We can also control

the attributes of the sphere, which make it more useful to use. We will name our sphere, “ball”, which will allow us to have several. We will also define the center, radius and color of the sphere. The nominal colors are red, blue, green, yellow, cyan, magenta, white and black. You can also specify the color using rgb with color=vector(r,g,b). If we run the following code snippet, we will see a cyan ball in the middle of the screen.

ball = sphere(pos=vector(0,0,0),radius=0.5,color=color.cyan)

A second object that is useful is the box. With the following commands, we will create a pair of boxes, which we call WallL and WallR, at the left and right sides of the screen. Running these commands will produce an image with a ball at the center of the screen, and walls on the left and right-hand sides. You can now explore the various zoom and perspective functions with this image.

WallL = box(pos=vector(-10,0,0),size=vector(0.1,10,5),color=color.blue)
WallR = box(pos=vector(10,0,0),size=vector(0.1,10,5),color=color.blue)

4.2 Animating the Ball

We would now like to animate the ball so that it can move across the screen. To do this, we will need to define new attributes of the ball. If it is going to move, we need a velocity, which we can set using the command below. At this point, we note that python knows nothing about units. We have defined positions and sizes as numbers. For simplicity, we assume that these are all in meters, but we need to manage the units. For a velocity, we will assume that the units are m/s, and we give the ball velocity \( \vec{v} = (1,0,0) \) m/s.

\[
\text{In } [2]: \quad \text{sphere()}
\]

\[
\text{In } [3]: \quad \text{from vpython import *}
\]

\[
\text{In } [4]: \quad \text{from math import *}
\]

\[
\text{In } [5]: \quad \text{ball = sphere(pos=vector(0,0,0),radius=0.5,color=color.cyan)}
\]

\[
\text{In } [6]: \quad \text{WallL = box(pos=vector(-10,0,0),size=vector(0.1,10,5),color=color.blue)}
\]

\[
\text{In } [7]: \quad \text{WallR = box(pos=vector(10,0,0),size=vector(0.1,10,5),color=color.blue)}
\]
ball.velocity = vector(1,0,0)

In addition to position and velocity, we will also need time (assumed to be in seconds). In animation, there are actually two times that may be relevant, the total elapsed time, and the time step between each animation frame. We need to set these as well. We will use $t$ for the total elapsed time and $dt$ for the time step. We initially set the elapsed time to 0 and the time step to be 0.1 s.

$t = 0$
$dt = 0.1$

We can now animate our ball. The underlying physics can be written as a difference equation:

$$\Delta \vec{r} = \vec{r} \Delta t$$

which we will implement using the logic that after one time step, the new position will be the old position plus the change in position. We also see that python is managing all the vectors for us. The following expression represents three equations, one for each of the $x$, $y$ and $z$ coordinates. We can just let python worry about that detail!

$$\vec{r}_{new} = \vec{r}_{old} + \Delta \vec{r}$$

When coding, we implement such an action using the expression

```
ball.pos = ball.pos + ball.velocity * dt
```

This leads to the following loop to animate the ball.

```
while True:
    ball.pos = ball.pos + ball.velocity*dt
    t = t + dt
```

Our code now looks like the following, and if we run this, the ball just goes off to the right forever. We run the code by clicking on the 
button at the top of the notebook. It knows nothing about walls, and we
have an infinite loop, so it never stops. *An object in motion remains in motion, except to the extent which it interacts with other objects.* We can force the program to stop by clicking on the stop symbol, as shown.

One other thing to note in the code below is the `rate(100)` that is in the loop. This limits the number of steps we take per real second, and without it our computer may run so fast that we see nothing. We now want to tell the ball how to interact with the walls. This is a simple elastic reflection that we need to consider. In a reflection, the component of the velocity that is normal to the reflecting surface changes sign. For us, this means that we need to reverse the $x$ component of velocity, or

$$v_x \rightarrow -v_x.$$

We also need to determine if the ball is interacting with the wall. We can approximate this by checking to see if the center of the ball would move to the opposite side of the wall. If it would, we apply the interaction. This can be done with the following conditional statements inside our animation loop.

```python
if ball.pos.x < WallL.pos.x:
    ball.velocity.x = -ball.velocity.x
if ball.pos.x > WallR.pos.x:
    ball.velocity.x = -ball.velocity.x
```

If we run this code, the ball will bounce back and forth between the two walls forever. We could change this by setting a finite time on the loop and stopping it after we have reached some total elapsed time.

```python
while t < 1000:
    rate(100)
    ball.pos = ball.pos + ball.velocity*dt
    t = t+dt
```

Because we have vectors, we can change the ball's velocity to be in more than one dimension. A simple change is to give the velocity a small $y$ component.

```python
ball.velocity = vector(1,0.1,0)
```

Running the code now, the ball will slowly move upward, bouncing between invisible walls as we neglected to code in the extent of the two walls.
4.3 Following the Object

Now that we have the ball bouncing around, we can introduce a couple of additional shapes to help us visualize the motion. The first shape is an *arrow* that we will use to indicate the direction of the ball. We need to initially create the arrow after we have defined both the ball and its velocity using the command.

```python
ball.bv = arrow(pos=ball.pos, axis=ball.velocity, color=ball.color)
```

The arrow then needs to be updated every time the ball moves, so at the end our loop we will add the line

```python
ball.bv.pos = ball.pos
ball.bv.axis = ball.velocity
```

The arrow also has a *length* attribute that we can change. The default is 1, but we may want to scale it to the actual length of the vector we are following. We would do that by setting the length of the arrow be equal to the magnitude of the velocity vector.

```python
ball.bv = arrow(pos=ball.pos, axis=ball.velocity, color=ball.color, length=mag(ball.velocity))
```

and

```python
ball.bv.pos = ball.pos
ball.bv.axis = ball.velocity
ball.bv.length = mag(ball.velocity)
```

We can also have the object leave a trail. There are a couple of ways to do this. The easiest is to use the *make_trail* attribute of the sphere when we created the ball. This will draw a trail that is the same color as the ball, and follows the ball forever.

```python
ball = sphere(pos=vector(0, 0, 0), radius=0.5, make_trail=True, color=color.cyan)
```

If we want more control over the trail, we can set additional attributes. We can either have a *curve* which is the default, or *points*. We can set the *interval* of how often a point is added, the default is every step, and we can set how many steps to retain on the trail using *retain*. Finally, it is possible to set the color.

```python
ball = sphere(pos=vector(0, 0, 0), radius=0.5, make_trail=True, color=color.cyan, interval=10, retain=250, trail_color=magenta)
```
There is an alternative way to leave a trail using the `curve` object. This has more attributes and functionality than the `make_trail`, so we describe it here. For most of our work in the course, they are 100% equivalent, especially if you have a finite trail length. The one place they differ is that the `make_trail` approach may have a glitch at the start of the trail. The `curve` will not. To use this method, we need to define the curve after we have defined the ball.

```python
ball = sphere(pos=vector(0,0,0),radius=0.5,color=color.cyan)
bball.trail = curve(color=color.magenta)
bball.trail.append(pos=bball.pos)
and then every time we move the ball, we need to append the position to the trail

```

```python
ball.pos = ball.pos + ball.velocity * dt
ball.trail.append(pos=ball.pos)
```

As a checkpoint, we show the current version of our code here where we have chosen to use the `make_trail` because we are only retaining 250 points along the path.

### 5 Additional Things to Do

Now that you have a working program, there are a few additional things to try.

First, add top and bottom walls, and make them touch, and then modify your code so the ball bounces off all four walls. If you want you can add `invisible` front and back walls as well.

Next add a constant acceleration to your program. The acceleration should update the velocity, and then the velocity will update the position.

Add a second ball bouncing around in the box. You can allow the two balls to interact if you would like.
6 References

- Vpython documentation can be found online at [https://vpython.org/contents/docs/index.html](https://vpython.org/contents/docs/index.html).
- Python documentation can be found online at [https://docs.python.org](https://docs.python.org).