Creating 2D graphs

OBJECTIVES
In this activity you will learn how to:
• Create and manipulate 2D graphs
• Create and display functions in graphs
• Use “for” loops

As a first step, add the following setup statements to the very top of your program:

```python
from visual import *
from visual.graph import *
```

These commands should always be present in any program that uses 2D graphics in VPython.

CONTINUOUS CURVES
In order to plot a 2D graph (for example, a functional dependence of position on time), we will be using two commands: one to create a graph object, and the other to actually plot the data points on that graph.

The setup command depends on the type of graph one would like to create. If a continuous curve is needed, then use command `gcurve`:

```python
gcurve(color=color.red)
```

where simultaneously we define the color of the curve. In practice, to use this curve - and to be able to add data points to it - one typically needs to name it. This is similar to naming other objects, such as naming a sphere with variable “ball”, so that VPython knows which sphere (or curve) you are referring to later in the program. For example, one can give it the name `func`:

```python
func = gcurve(color=color.red)
```

Once the curve is set up, there is nothing on the plot yet, and one needs to add data points one by one.

To add a point to your graph, refer to the graph by its name and use attribute `plot`:

```python
func.plot(pos=(x,y))
```

Here `x` and `y` are the coordinates of the point you would like to add to the curve. The `y` coordinate may be a function of `x` [for example, `pos=(x, exp(x))`] or both coordinates may be functions of some third variable [for example, `pos=(cos(t), sin(t))`]. The two coordinates may also be completely unrelated.

It is important to emphasize that the `plot` command above adds ONLY ONE point to a curve. So, you actually need to repeat this command many times to create the actual curve, adding points for
different values of the argument. To do so, we will typically use a loop (cycle). Here is an example that uses a `while` loop we are already familiar with:

```python
func = gcurve(color=color.red)
x = 0
dx = 0.01
while x < 1:
    func.plot(pos=(x,exp(x)))
    x = x + dx
```

The above example plots function $\exp(x)$ in the range of values of variable $x$ from 0 to 1. Run this program and examine the result (do not forget the two setup statements `from` at the beginning of your program).

Another alternative is to use a `for` loop. The format of this command explicitly assigns values to a loop variable, so that you do not have to assign the initial and final values as well as the increment anywhere other than in the loop command itself:

```python
for <variable> in arange(<initial_value>,<final_value>,<increment>):

for example,

```python
for x in arange(0,10,0.1):
```

Here, variable $x$ will be assigned values from 0 to 10 with increment 0.1 (i.e. 0, 0.1, 0.2,..., 9.9, 10) on each run of the body of the loop. The body of the loop follows after the colon and must be indented to the right. The loop stops after it has run with all the allowed values of the loop variable (e.g., 101 times in the example above).

Here is an example of the use of the `for` loop to plot a continuous curve, where we also illustrate the use of a parameter variable $t$ in both horizontal and vertical coordinates of the curve points:

```python
func = gcurve(color=color.blue)
for t in arange(0,2*pi,2*pi/100):
    func.plot(pos=(cos(t),sin(t)))
```

Run this program and examine the result. How would you modify this program to plot only the left half of the circle? What values do you need to change? What needs to be changed to change the radius of the circle? Note that the circle will appear elliptical on your graph unless you change the default ratio of the graph’s horizontal width to its vertical height (aspect ratio). See below for how to make such a change using the `gdisplay` command.

Note that in order to produce a smooth curve the step (increment) in the loop variable in both types of the loops must be small compared to the range over which you vary your variable ($x$ in the first example and $t$ in the second example). Note also that common trigonometric functions in VPython assume an argument in radians, not degrees.
DISCRETE PLOTS
VPython also provides tools for creating discrete plots, i.e. plots that consist of only data points, without connecting the dots into a curve. To setup an object of a discrete plot, use command `gdots`:

```python
gdots(color=color.red)
```

Modify any of the two examples above by replacing `gcurve` with `gdots` and examine the result.

SEVERAL PLOTS ON THE SAME GRAPH
You can plot several functions within the same graph window. First, you need to set up different curves with different names and preferably different colors. For example, for two plots on the same graph, the two curves can be set up the following way:

```python
func1 = gcurve(color=color.red)
func2 = gcurve(color=color.blue)
```

(Use `gdots` if you need discrete plots). Then, add data points to each of the curves by using the respective curve names with attribute `plot`:

```python
func1.plot(pos=(x1,y1))
func2.plot(pos=(x2,y2))
```

In most practical cases, these commands will need to be located inside a cycle/loop the same way as we did for a single curve.

CHECKING YOUR UNDERSTANDING
Write a program that plots two functions $f(x) = 10 \cos(x) \exp(-x/10)$ and $g(x) = 10 \sin(x) \exp(-x/10)$ within the same window in the range of $x$ from 0 to 20. One function should be plotted as a continuous curve, and the other should be plotted as discrete points. Use different colors. This exercise should be repeated TWICE: first with a `while` loop, and second with a `for` loop. Note that you do not need to create two different loops for each of the two graphs since the range of the $x$ variable is the same for both graphs. Your result should be checked against the plot below.
PLOTTING GRAPHS IN DIFFERENT WINDOWS

Up until this point we discussed the plots within the same window. However, for many physical situations we may want to create different graph windows for different functions. For example, if we would like to plot both position vs. time and momentum vs. time for a certain physical system, we should not plot them in the same window, since position and momentum have different dimensionality and different physical units. Hence we need to be able to produce different windows and to control their parameters, such as their position, size, window titles, axis titles, axis ranges, and background and foreground colors. All these tasks are achieved by the command `gdisplay`. Different windows are called “graphic displays” in VPython.

First, one needs to initialize graphic displays (also setting values of all parameters if necessary) and then use them for each of the graphs. To initialize graphic displays, use the following format:

```python
gdisplay(x=<window_x-coordinate>, y=<window_y-coordinate>,
width=<window_width>, height=<window_height>, title='window_title',
xtitle='horizontal_axis_title', ytitle='vertical_axis_title',
xmin=<minimal_value_of_x>, xmax=<maximal_value_of_x>,
ymin=<minimal_value_of_y>, ymax=<maximal_value_of_y>,
foreground=<foreground_color>, background=<background_color>)
```

The first two parameters set the position of the upper left corner of the window (in pixels) with respect to the upper left corner of the computer display (which has coordinates (0,0)). The width and the height of the window are measured in pixels as well. The titles of the window and both axes can be represented by any text enclosed in single quotation marks. Instead of autoscaling to show the entire graph, you can set the limits on the horizontal and vertical coordinates displayed within each window (parameters `xmin`, `xmax`, `ymin`, and `ymax`). Finally, the colors are set the standard way (`color.whatever`).

Some or all parameters are allowed to be missing. The default values are `x=0`, `y=0`, `width=800`, `height=400`, no titles, fully autoscaled, white foreground and black background.

In order to use various graphic displays, you need to assign them some names, for example,

```python
display1 = gdisplay(x=0, y=0, width=600, height=300)
display2 = gdisplay(x=0, y=300, width=600, height=300)
```

and then refer to them by name when initializing various graph curves:

```python
function1 = gcurve(gdisplay=display1, color=color.red)
function2 = gcurve(gdisplay=display1, color=color.blue)
function3 = gdots(gdisplay=display2, color=color.red)
function4 = gdots(gdisplay=display2, color=color.blue)
```

As a result of these definitions, the VPython program will display two windows and plot the first and the second graphs in the first window (as red and blue continuous curves) and the third and the fourth curves in the second window (as red and blue discrete plots). Now plot four different functions using these definitions and examine the result.