Engineering Computations

Module 3: Fly at change in systems

Lorena A. Barba and Natalia C. Clementi

© 2017
Content under Creative Commons Attribution license CC-BY 4.0, code under BSD 3-Clause License. Find source Jupyter notebooks at https://github.com/engineersCode/EngComp

Lesson 1: Catch things in motion

This module of the *Engineering Computations* course is our launching pad to investigate *change*, *motion*, *dynamics*, using computational thinking, Python, and Jupyter.

The foundation of physics and engineering is the subject of **mechanics**: how things move around, when pushed around. Or pulled… in the beginning of the history of mechanics, Galileo and Newton seeked to understand how and why objects fall under the pull of gravity.

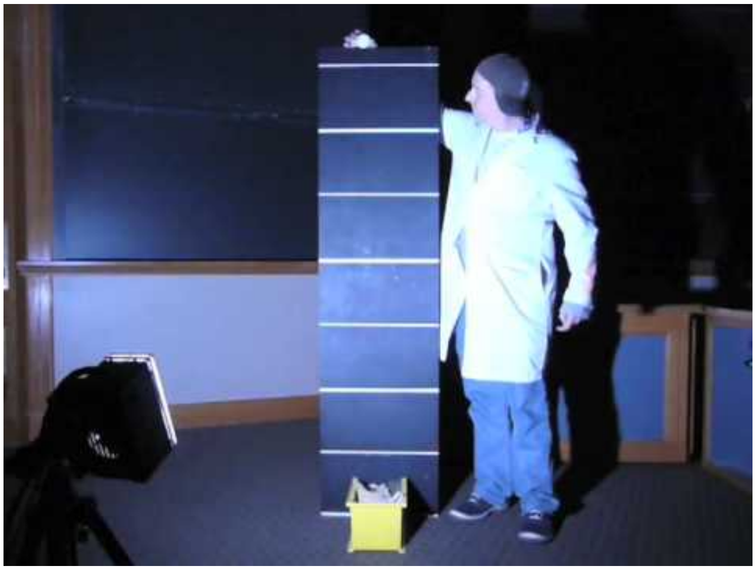
This first lesson will explore motion by analyzing images and video, to learn about velocity and acceleration.

1 Acceleration of a falling ball

Let's start at the beginning. Suppose you want to use video capture of a falling ball to *compute* the acceleration of gravity. Could you do it? With Python, of course you can!

Here is a neat video we found online, produced over at MIT several years ago [1]. It shows a ball being dropped in front of a metered panel, while lit by a stroboscopic light. Watch the video!

```
In [1]: from IPython.display import YouTubeVideo
        vid = YouTubeVideo("xQ4znShlK5A")
        display(vid)
```



We learn from the video that the marks on the panel are every 0.25m, and on the website they say that the strobe light flashes at about 15 Hz (that's 15 times per second). The final image on Flickr, however, notes that the strobe fired 16.8 times per second. So we have some uncertainty already!

Luckily, the MIT team obtained one frame with the ball visible at several positions as it falls. This, thanks to the strobe light and a long-enough exposure of that frame. What we'd like to do is use that frame to capture the ball positions digitally, and then obtain the velocity and acceleration from the distance over time.

You can find several toolkits for handling images and video with Python; we'll start with a simple one called `imageio`. Import this library like any other, and let's load `numpy` and `pyplot` while we're at it.

```
In [2]: import imageio
        import numpy
        from matplotlib import pyplot
```

1.1 Read the video

With the `get_reader()` method of `imageio`, you can read a video from its source into a *Reader* object. You don't need to worry too much about the technicalities here—we'll walk you through it all—but check the type, the length (for a video, that's number of frames), and notice you can get info, like the frames-per-second, using `get_meta_data()`.

```
In [3]: reader = imageio.get_reader(
          'http://techtv.mit.edu/videos/831-strobe-of-a-falling-ball/download.mp4')
```

```
In [4]: type(reader)
```

```
Out[4]: imageio.plugins.ffmpeg.FfmpegFormat.Reader
```

```
In [5]: len(reader)
```

```
Out[5]: 1235
```

```
In [6]: fps = reader.get_meta_data()['fps']
        print(fps)
```

```
30.0
```

Note: You may get this error after calling `get_reader()`:

NeedDownloadError: Need ffmpeg exe. You can obtain it with either:

- install using conda: `conda install ffmpeg -c conda-forge`
- download by calling: `imageio.plugins.ffmpeg.download()`

If you do, follow the tips to install the needed `ffmpeg` tool.

1.2 Show a video frame in an interactive figure

With `imageio`, you can grab one frame of the video, and then use `pyplot` to show it as an image. But we want to interact with the image, somehow.

So far in this course, we have used the command `%matplotlib inline` to get our plots rendered *inline* in a Jupyter notebook. There is an alternative command that gives you some interactivity on the figures: `%matplotlib notebook`. Execute this now, and you'll see what it does below, when you show the image in a new figure.

Let's also set some font parameters for our plots in this notebook.

```
In [7]: %matplotlib notebook

        #Import rcParams to set font styles
        from matplotlib import rcParams

        #Set font style and size
        rcParams['font.family'] = 'serif'
        rcParams['font.size'] = 16
```

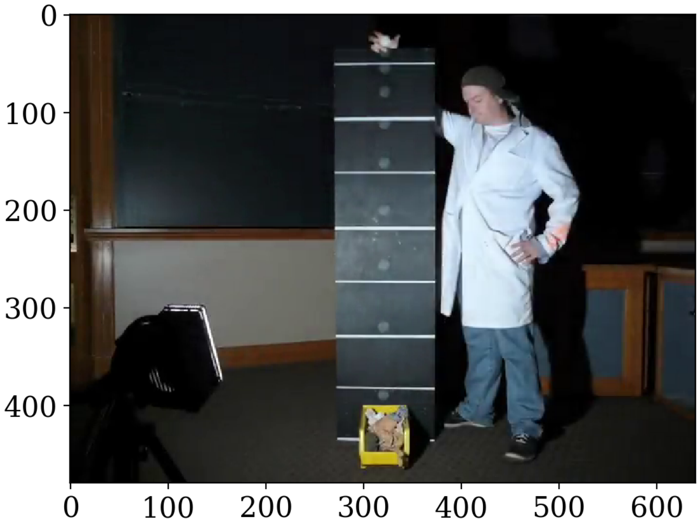
Now we can use the `get_data()` method on the `imageio` *Reader* object, to grab one of the video frames, passing the frame number. Below, we use it to grab frame number 1100, and then print the `shape` attribute to see that it's an "array-like" object with three dimensions: they are the pixel numbers in the horizontal and vertical directions, and the number of colors (3 colors in RGB format). Check the type to see that it's an `imageio` *Image* object.

```
In [8]: image = reader.get_data(1100)
        image.shape

Out[8]: (480, 640, 3)

In [9]: type(image)

Out[9]: imageio.core.util.Image
```

Naturally, `imageio` plays well with `pyplot`. You can use `pyplot.imshow()` to show the image in a figure. We chose to show frame 1100 after playing around a bit and finding that it gives a good view of the long-exposure image of the falling ball.

Explore: Check out the neat interactive options that we get with `%matplotlib notebook`. Then go back and change the frame number above, and show it below. Notice that you can see the (x, y) coordinates of your cursor tip while you hover on the image with the mouse.

```
In [10]: pyplot.imshow(image, interpolation='nearest');
```



1.3 Capture mouse clicks on the frame

Okay! Here is where things get really interesting. Matplotlib has the ability to create event connections, that is, connect the figure canvas to user-interface events on it, like mouse clicks.

To use this ability, you write a function with the events you want to capture, and then connect this function to the Matplotlib "event manager" using `mpl_connect()`. In this case, we connect the `'button_press_event'` to the function named `onclick()`, which captures the (x, y) coordinates of the mouse click on the figure. Magic!

```
In [11]: fig = pyplot.figure()

         pyplot.imshow(image, interpolation='nearest')

         coords = []
         def onclick(event):
             '''Capture the x,y coordinates of a mouse click on the image'''
             ix, iy = event.xdata, event.ydata
             coords.append([ix, iy])

         connectId = fig.canvas.mpl_connect('button_press_event', onclick)
```

```
<IPython.core.display.Javascript object>
<IPython.core.display.HTML object>
```

Notice that in the code cell above, we created an empty list named `coords`, and inside the `onclick()` function, we are appending to it the (x, y) coordinates of each mouse click on the figure. After executing the cell above, you have a connection to the figure, via the user interface: try clicking with your mouse on the endpoints of the white lines of the metered panel (click on the edge of the panel to get approximately equal x coordinates), then print the contents of the `coords` list below.

```
In [12]: coords
```

```
Out[12]: [[270.77840909090912, 53.306818181818073],
          [270.77840909090912, 107.85227272727263],
          [272.07711038961043, 163.6964285714285],
          [272.07711038961043, 219.54058441558436],
          [272.07711038961043, 274.08603896103892],
          [272.07711038961043, 328.63149350649348],
          [273.37581168831173, 383.17694805194799],
          [274.67451298701303, 435.125]]
```

The x coordinates are pretty close, but there is some variation due to our shaky hand (or bad eyesight), and perhaps because the metered panel is not perfectly vertical. We can cast the `coords` list to a NumPy array, then grab all the first elements of the coordinate pairs, then get the standard deviation as an indication of our error in the mouse-click captures.

```
In [13]: numpy.array(coords)[:,0]
```

```
Out[13]: array([ 270.77840909,  270.77840909,  272.07711039,  272.07711039,
                 272.07711039,  272.07711039,  273.37581169,  274.67451299])
```

```
In [14]: numpy.array(coords)[:,0].std()
```

```
Out[14]: 1.2039283258272222
```

Depending how shaky *your* hand was, you may get a different value, but we got a standard deviation of about one pixel. Pretty good!

Now, let's grab all the second elements of the coordinate pairs, corresponding to the y coordinates, i.e., the vertical positions of the white lines on the video frame.

```
In [15]: y_lines = numpy.array(coords)[:,1]
         y_lines
```

```
Out[15]: array([  53.30681818,  107.85227273,  163.69642857,  219.54058442,
                 274.08603896,  328.63149351,  383.17694805,  435.125     ])
```

Looking ahead, what we'll do is repeat the process of capturing mouse clicks on the image, but clicking on the ball positions. Then, we will want to have the vertical positions converted to physical length (in meters), from the pixel numbers on the image.

You can get the scaling from pixels to meters via the distance between two white lines on the metered panel, which we know is 0.25m.

Let's get the average vertical distance between two while lines, which we can calculate as:

$$\overline{\Delta y} = \sum_{i=0}^{N} \frac{y_{i+1} - y_i}{N-1} \tag{1}$$

```
In [16]: gap_lines = y_lines[1:] - y_lines[0:-1]
         gap_lines.mean()
```

```
Out[16]: 54.545454545454561
```

Discuss with your neighbor

- Why did we slice the `y_lines` array like that? If you can't explain it, write out the first few terms of the sum above and think!

1.4 Compute the acceleration of gravity

We're making good progress! You'll repeat the process of showing the image on an interactive figure, and capturing the mouse clicks on the figure canvas: but this time, you'll click on the ball positions.

Using the vertical displacements of the ball, Δy_i, and the known time between two flashes of the strobe light, 1/16.8s, you can get the velocity and acceleration of the ball! But first, to convert the vertical displacements to meters, you'll multiply by 0.25m and divide by `gap_lines.mean()`.

Before clicking on the ball positions, you may want to inspect the high-resolution final photograph on Flickr—notice that the first faint image of the falling ball is just "touching" the ring finger of Bill's hand. We decided *not* to use that photograph in our lesson because the Flickr post says *"All rights reserved"*, while the video says specifically that it is licensed under a Creative Commons license. In other words, MIT has granted permission to use the video, but *not* the photograph. *Sigh.*

OK. Go for it: capture the clicks on the ball!

```
In [17]: fig = pyplot.figure()

         pyplot.imshow(image, interpolation='nearest')

         coords = []
         def onclick(event):
             '''Capture the x,y coordinates of a mouse click on the image'''
             ix, iy = event.xdata, event.ydata
             coords.append([ix, iy])

         connectId = fig.canvas.mpl_connect('button_press_event', onclick)
```

```
<IPython.core.display.Javascript object>
<IPython.core.display.HTML object>
```

```
In [18]: coords
```

```
Out[18]: [[321.42775974025972, 38.249999999999886],
          [321.42775974025972, 55.13311688311677],
          [321.42775974025972, 81.107142857142776],
          [321.42775974025972, 112.27597402597394],
          [321.42775974025972, 152.53571428571422],
          [321.42775974025972, 203.18506493506487],
          [320.12905844155841, 257.73051948051943],
          [320.12905844155841, 321.36688311688306],
          [320.12905844155841, 390.1980519480519]]
```

We'll scale the vertical displacements of the falling ball as explained above (to get distance in meters), then use the known time between flashes of the strobe light, 1/16.8s, to compute estimates of the velocity and acceleration of the ball at every captured instant, using:

$$v_i = \frac{y_{i+1} - y_i}{\Delta t}, \qquad a_i = \frac{v_{i+1} - v_i}{\Delta t} \tag{2}$$

```
In [19]: y_coords = numpy.array(coords)[:,1]
         delta_y = (y_coords[1:] - y_coords[:-1]) *0.25 / gap_lines.mean()
```

```
In [20]: v = delta_y * 16.8
         v
```

```
Out[20]: array([ 1.3,  2. ,  2.4,  3.1,  3.9,  4.2,  4.9,  5.3])
```

```
In [21]: a = (v[1:] - v[:-1]) *16.8
         a
```

```
Out[21]: array([ 11.76,   6.72,  11.76,  13.44,   5.04,  11.76,   6.72])
```

```
In [22]: a[1:].mean()
```

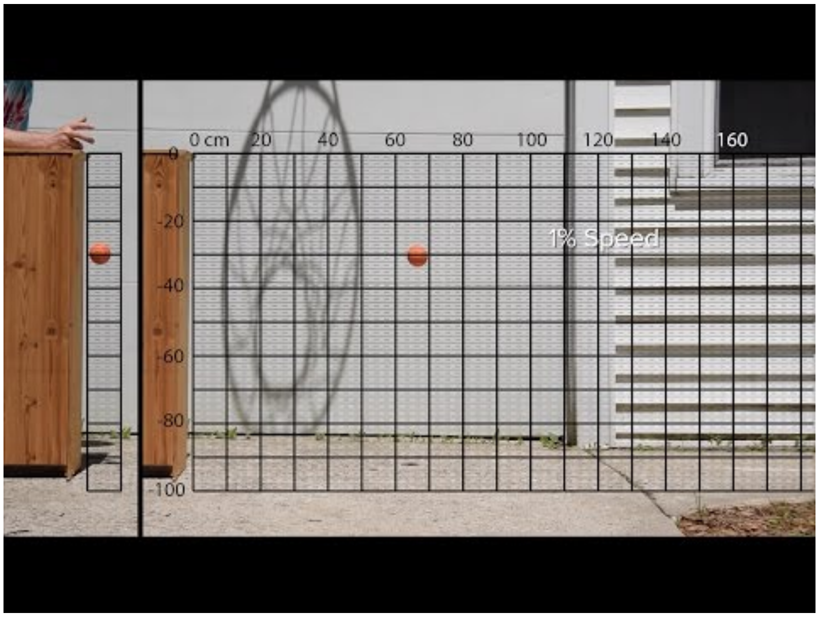
```
Out[22]: 9.2399999999999913
```

Yikes! That's some wide variation on the acceleration estimates. Our average measurement for the acceleration of gravity is not great, but it's not far off… In case you don't remember, the actual value is 9.8m/s^2.

2 Projectile motion

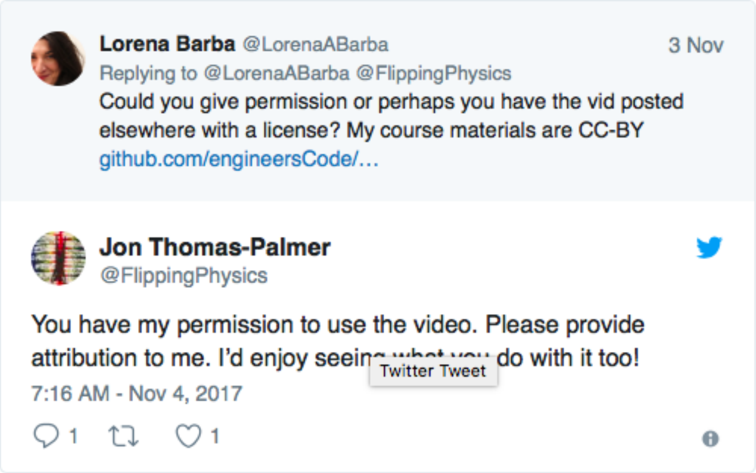
Now, we'll study projectile motion, using a video of a ball "fired" horizontally, like a projectile. Here's a neat video we found online, produced by the folks over at Flipping Physics [2].

```
In [23]: from IPython.display import YouTubeVideo
         vid = YouTubeVideo("Y4jgJK35Gf0")
         display(vid)
```



We used Twitter to communicate with the author of the video and ask permission to use it in this lesson. A big *Thank You* to Jon Thomas-Palmer for letting us use it!

```
In [24]: %%html
         <blockquote class="twitter-tweet" data-lang="en"><p lang="en" dir="ltr"> ...
```



Lorena Barba @LorenaABarba 3 Nov
Replying to @LorenaABarba @FlippingPhysics
Could you give permission or perhaps you have the vid posted elsewhere with a license? My course materials are CC-BY
github.com/engineersCode/…

Jon Thomas-Palmer
@FlippingPhysics

You have my permission to use the video. Please provide attribution to me. I'd enjoy seein do with it too!
Twitter Tweet
7:16 AM - Nov 4, 2017

2.1 Capture mouse clicks on many frames with a widget

We'd like to capture the coordinates of mouse clicks on a *sequence* of images, so that we may have the positions of the moving ball caught on video. We know how to capture the coordinates of mouse clicks, so the next challenge is to get consecutive frames of the video displayed for us, to click on the ball position each time.

Widgets to the rescue! There are currently 10 different widget types included in the `ipywidgets` library. The `BoundedIntText()` widget shows a text box with an integer value that can be stepped from a minimum to a maximum value by clicking up/down arrows. Stepping through frames with this widget, and clicking on the ball position each time, gets us what we want.

Digitizing the ball positions in this way is a bit tedious. But this could be a realistic scenario: you captured video of a moving object, and you need to get position data from the video frames. Unless you have some fancy motion-capture equipment, this will do the trick.

Let's load the Jupyter widgets:

```
In [25]: from ipywidgets import widgets
```

Download the video, previously converted to .mp4 format to be read by `imageio`, and then load it to an `imageio` *Reader*. Notice that it has 3531 frames, and they are 720x1280 pixels in size.

Below, we're showing frame number 52, which we found to be the start of the portion shown at 50% speed. Go ahead and use that frame to capture mouse clicks on the intersection of several 10cm lines with one vertical, so you can calculate the scaling from pixels to physical distance.

```
In [26]: from urllib.request import urlretrieve
         URL = 'http://go.gwu.edu/engcomp3vid1?accessType=DOWNLOAD'
         urlretrieve(URL, 'Projectile_Motion.mp4')

Out[26]: ('Projectile_Motion.mp4', <http.client.HTTPMessage at 0x116a211d0>)

In [27]: reader = imageio.get_reader('Projectile_Motion.mp4')

In [28]: len(reader)

Out[28]: 3531

In [29]: image = reader.get_data(52)
         image.shape

Out[29]: (720, 1280, 3)

In [30]: fig = pyplot.figure()
         pyplot.imshow(image, interpolation='nearest')

         coords2 = []
         def onclick(event):
             '''Capture the x,y coordinates of a mouse click on the image'''
             ix, iy = event.xdata, event.ydata
             coords2.append([ix, iy])

         connectId = fig.canvas.mpl_connect('button_press_event', onclick)

<IPython.core.display.Javascript object>
<IPython.core.display.HTML object>


In [31]: coords2

Out[31]: [[723.33064516129025, 117.98387096774195],
          [723.33064516129025, 169.59677419354841],
          [723.33064516129025, 221.20967741935488],
          [723.33064516129025, 275.40322580645159],
          [723.33064516129025, 332.17741935483866],
          [723.33064516129025, 381.20967741935476],
          [723.33064516129025, 437.98387096774195],
          [723.33064516129025, 489.59677419354836],
          [723.33064516129025, 543.79032258064512]]
```

```
In [32]: y_lines2 = numpy.array(coords2)[:,1]
         y_lines2
```

```
Out[32]: array([ 117.98387097,  169.59677419,  221.20967742,  275.40322581,
                 332.17741935,  381.20967742,  437.98387097,  489.59677419,
                 543.79032258])
```

```
In [33]: gap_lines2 = y_lines2[1:] - y_lines2[0:-1]
         gap_lines2.mean()
```

```
Out[33]: 53.225806451612897
```

Above, we repeated the process to compute the vertical distance between the 10cm marks (averaging over our clicks): the scaling of distances from this video will need multiplying by 0.1 to get meters, and dividing by `gap_lines2.mean()`.

Now the fun part! Study the code below: we create a `selector` widget of the `BoundedIntText` type, taking the values from 52 to 77, and stepping by 1. We already played around a lot with the video and found this frame range to contain the portion shown at 50% speed.

We re-use the `onclick()` function, this time appending to a list named `coords2`, and we call it with an event connection from Matplotlib, just like before. But now we add a call to `widgets.interact()`, using a new function named `catchclick()` that reads a new video frame and refreshes the figure with it.

Execute this cell, then click on the ball position, advance a frame, click on the new ball position, and so on, until frame 77. The mouse click positions will be saved in `coords2`.

We found it better to click on the bottom edge of the ball image, rather than attempt to aim at the ball's center.

```
In [34]: selector = widgets.BoundedIntText(value=52, min=52, max=77, step=1,
             description='Frame:',
             disabled=False)

         coords2 = []
         def onclick(event):
             '''Capture the x,y coordinates of a mouse click on the image'''
             ix, iy = event.xdata, event.ydata
             coords2.append([ix, iy])

         def catchclick(frame):
             image = reader.get_data(frame)
             pyplot.imshow(image, interpolation='nearest');
         fig = pyplot.figure()

         connectId = fig.canvas.mpl_connect('button_press_event', onclick)

         widgets.interact(catchclick, frame=selector);
```

```
<IPython.core.display.Javascript object>
<IPython.core.display.HTML object>
```

```
A Jupyter Widget
```

```
In [35]: coords2
```

```
Out[35]: [[297.52419354838707, 117.82258064516134],
          [320.74999999999994, 117.82258064516134],
          [343.97580645161287, 120.40322580645159],
          [367.20161290322579, 125.56451612903231],
          [393.00806451612908, 130.72580645161293],
          [416.23387096774189, 138.4677419354839],
          [439.45967741935482, 151.37096774193549],
          [462.68548387096774, 159.11290322580646],
          [485.91129032258067, 172.01612903225805],
          [511.71774193548384, 182.33870967741939],
          [532.36290322580635, 197.82258064516134],
          [558.16935483870975, 218.4677419354839],
          [581.39516129032268, 233.95161290322585],
          [602.04032258064512, 252.01612903225805],
          [625.26612903225805, 272.66129032258061],
          [651.07258064516122, 295.88709677419354],
          [676.8790322580644, 319.11290322580646],
          [697.52419354838707, 344.91935483870964],
          [723.33064516129025, 373.30645161290317],
          [746.55645161290317, 399.11290322580646],
          [767.20161290322585, 430.08064516129025],
          [790.42741935483878, 461.04838709677415],
          [816.23387096774195, 494.59677419354836],
          [839.45967741935488, 528.14516129032245],
          [862.68548387096757, 566.85483870967732],
          [888.49193548387098, 600.40322580645147]]
```

Now, convert the positions in pixels to meters, using our scaling for this video, and save the *x* and *y* coordinates to new arrays. Below, we plot the ball positions that we captured.

```
In [36]: x = numpy.array(coords2)[:,0] *0.1 / gap_lines2.mean()
         y = numpy.array(coords2)[:,1] *0.1 / gap_lines2.mean()
```

```
In [37]: fig = pyplot.figure()
         pyplot.scatter(x,-y);
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

Finally, compute the vertical displacements, then get the vertical velocity and acceleration. And why not repeat the process for the horizontal direction of motion. The time interval is 1/60 seconds, according to the original video description, i.e., 60 frames per second.

```
In [38]: delta_y = (y[1:] - y[:-1])
```

```
In [39]: vy = delta_y * 60
         ay = (vy[1:] - vy[:-1]) * 60
         print('The acceleration in the y direction is: {:.2f}'.format(ay.mean()))
```

```
The acceleration in the y direction is: 9.45
```

```
In [40]: delta_x = (x[1:] - x[:-1])
         vx = delta_x * 60
         ax = (vx[1:] - vx[:-1]) * 60
         print('The acceleration in the x direction is: {:.2f}'.format(ax.mean()))
```

```
The acceleration in the x direction is: 0.73
```

Discuss

- What did you get for the x and y accelerations? What did your neighbor get?
- Do the results make sense to you? Why or why not?

3 Numerical derivatives

What we did above to compute the average velocity between two captured ball positions is equivalent to a *numerical derivative*. The velocity is the *derivative* of position with respect to time, and we can approximate its instantaneous value with the average velocity between two close instants in time:

$$v(t) = \frac{dy}{dt} \approx \frac{y(t_i + \Delta t) - y(t_i)}{\Delta t} \tag{3}$$

And acceleration is the *derivative* of velocity with respect to time; we can approximate it with the average acceleration within a time interval:

$$a(t) = \frac{dv}{dt} \approx \frac{v(t_i + \Delta t) - v(t_i)}{\Delta t} \tag{4}$$

As you can imagine, the quality of the approximation depends on the size of the time interval: as Δt gets smaller, the error also gets smaller.

3.1 Using high-resolution data

Suppose we had some high-resolution experimental data of a falling ball. Might we be able to compute the acceleration of gravity, and get a value closer to the actual acceleration of $9.8\mathrm{m/s}^2$?

You're in luck! Physics professor Anders Malthe-Sørenssen of Norway has some high-resolution data on the website to accompany his book [3]. We contacted him by email to ask for permission to use the data set of a falling tennis ball, and he graciously agreed. *Thank you!* His data was recorded with a motion detector on the ball, measuring the y coordinate at tiny time intervals of $\Delta t = 0.001\mathrm{s}$. Pretty fancy.

We have the data in our repository for this course, but you may have to download it first, if you got this Jupyter notebook by itself. If so, add a code cell below, and execute:

```
filename = 'fallingtennisball02.txt'
url = 'http://go.gwu.edu/engcomp3data1'
urlretrieve(url, filename)
```

You already imported `urlretrieve` above to get the video. Remember to then comment the assignment of the `filename` variable below.

```
In [41]: filename = '../../data/fallingtennisball02.txt'
         t, y = numpy.loadtxt(filename, usecols=[0,1], unpack=True)
```

Okay! We should have two new arrays with the time and position data. Let's get a plot of the ball's vertical position.

```
In [42]: fig = pyplot.figure(figsize=(6,4))
         pyplot.plot(t,y);
```

```
<IPython.core.display.Javascript object>
<IPython.core.display.HTML object>
```

Neat. The ball bounced 3 times during motion capture. Let's compute the acceleration during the first fall, before the bounce. A quick check on the `y` array shows that there are several points that take a negative value. We can use the first negative entry as the top limit of a slice, and then compute displacements, velocity, and acceleration with that slice.

```
In [43]: numpy.where( y < 0 )[0]
```

```
Out[43]: array([ 576,  577,  578,  579,  580,  581,  582,  583,  584,  585, 1420,
                1421, 1422, 1423, 1424, 1425, 1426, 1427, 1428, 1429, 2048, 2049,
                2050, 2051, 2052, 2053, 2054, 2055, 2056, 2057])
```

```
In [44]: delta_y = (y[1:576] - y[:575])
```

```
In [45]: dt = t[1]-t[0]
         dt
```

```
Out[45]: 0.0010000000000000002
```

```
In [46]: vy = delta_y / dt
         ay = (vy[1:] - vy[:-1]) / dt
         print('The acceleration in the y direction is: {:.2f}'.format(ay.mean()))
```

```
The acceleration in the y direction is: -9.55
```

Gah. Even with this high-resolution data, we're getting an average value of acceleration that is smaller than the actual acceleration of gravity: 9.8m/s^2. *What is going on?* Hmm. Let's make a plot of the acceleration values…

```
In [47]: fig = pyplot.figure(figsize=(6,4))
         pyplot.plot(ay);
```

```
<IPython.core.display.Javascript object>
<IPython.core.display.HTML object>
```

Discuss with your neighbor

- What do you see in the plot of acceleration computed from the high-resolution data?
- Can you explain it? What do you think is causing this?

4 What we've learned

- Work with images and videos in Python using `imageio`.
- Get interactive figures using the `%matplotlib notebook` command.
- Capture mouse clicks with Matplotlib's `mpl_connect()`.
- Observed acceleration of falling bodies is less than 9.8m/s^2.
- Capture mouse clicks on several video frames using widgets!
- Projectile motion is like falling under gravity, plus a horizontal velocity.
- Compute numerical derivatives using differences via array slicing.
- Real data shows free-fall acceleration decreases in magnitude from 9.8m/s^2.

5 References

1. Strobe of a Falling Ball (2008), MIT Department of Physics Technical Services Group, video under CC-BY-NC, available online on MIT TechTV.

2. The Classic Bullet Projectile Motion Experiment with X & Y Axis Scales (2004), video by Flipping Physics, Jon Thomas-Palmer. Used with permission.

3. *Elementary Mechanics Using Python* (2015), Anders Malthe-Sorenssen, Undergraduate Lecture Notes in Physics, Springer. Data at http://folk.uio.no/malthe/mechbook/

Lesson 2: Step to the future

Welcome to Lesson 2 of the course module "Fly at changing systems," in *Engineering Computations*. The previous lesson, Catch things in motion, showed you how to compute velocity and acceleration of a moving body whose positions were known.

Time history of position can be captured on a long-exposure photograph (using a strobe light), or on video. But digitizing the positions from images can be a bit tedious, and error-prone. Luckily, we found online a data set from a fancy motion-capture experiment of a falling ball, with high resolution [1]. You computed acceleration and found that it was not only smaller than the theoretical value of $9.8\mathrm{m/s}^2$, but it *decreased* over time. The effect is due to air resistance and is what leads to objects reaching a *terminal velocity* in freefall.

In general, not only is motion capture (a.k.a., *mo-cap*) expensive, but it's inappropriate for many physical scenarios. Take a roller-coaster ride, for example: during design of the ride, it's more likely that the engineers will use an *accelerometer*. It really is the acceleration that makes a roller-coaster ride exciting, and they only rarely go faster than highway speeds (say, 60 mph) [2]. How would an engineer analyze data captured with an accelerometer?

1 A roller-coaster ride

Prof. Anders Malthe-Sorenssen has a file with accelerometer data for a roller-coaster ride called "The Rocket" (we don't know if it's real or made up!). He has kindly given permission to use his data. So let's load it and have a look. We'll first need our favorite numerical Python libraries, of course.

```
In [1]: import numpy
        from matplotlib import pyplot
```

```
In [2]: %matplotlib inline

        #Import rcParams to set font styles
        from matplotlib import rcParams

        #Set font style and size
        rcParams['font.family'] = 'serif'
        rcParams['font.size'] = 14
```

If you don't have the data file in the location we assume below, you can get it by adding a code cell and executing this code in it, then commenting or deleting the `filename` assignment before the call to `numpy.loadtxt()`.

```
from urllib.request import urlretrieve
URL = 'http://go.gwu.edu/engcomp3data2?accessType=DOWNLOAD'
urlretrieve(URL, 'therocket.txt')
```

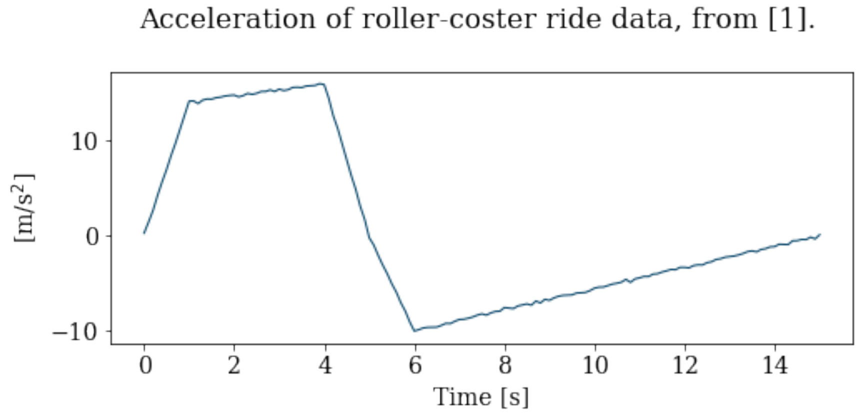
```
In [3]: filename = '../../data/therocket.txt'
        t, a = numpy.loadtxt(filename, usecols=[0,1], unpack=True)
```

We'll take a peek at the data by printing the first five pairs of (t, a) values, then plot the whole set below. Time is given in units of seconds, while acceleration is in $\mathrm{m/s}^2$.

```
In [4]: for i in range(0,5): print(t[i],a[i])
```

```
0.0 0.2731644
0.1 1.4411079
0.2 2.6693138
0.3 4.2383806
0.4 5.6499504
```

```
In [5]: fig = pyplot.figure(figsize=(8, 3))

        pyplot.plot(t, a, color='#004065', linestyle='-', linewidth=1)
        pyplot.title('Acceleration of roller-coster ride data, from [1]. \n')
        pyplot.xlabel('Time [s]')
        pyplot.ylabel('[m/s$^2$]');
```



Our challenge now is to find the motion description—the position $x(t)$—from the acceleration data. In the previous lesson, we did the opposite: with position data, get the velocity and acceleration, using *numerical derivatives*:

$$v(t_i) = \frac{dx}{dt} \approx \frac{x(t_i + \Delta t) - x(t_i)}{\Delta t} \tag{1}$$

$$a(t_i) = \frac{dv}{dt} \approx \frac{v(t_i + \Delta t) - v(t_i)}{\Delta t} \tag{2}$$

Since this time we're dealing with horizontal acceleration, we swapped the position variable from y to x in the equation for velocity, above.

The key to our problem is realizing that if we have the initial velocity, we can use the acceleration data to find the velocity after a short interval of time. And if we have the initial position, we can use the known velocity to find the new position after a short interval of time. Let's rearrange the equation for acceleration above, by solving for the velocity at $t_i + \Delta t$:

$$v(t_i + \Delta t) \approx v(t_i) + a(t_i)\Delta t \tag{3}$$

We need to know the velocity and acceleration at some initial time, t_0, and then we can compute the velocity $v(t_i + \Delta t)$. For the roller-coaster ride, it's natural to assume that the initial velocity is zero, and the initial position is zero with respect to a convenient reference system. We're actually ready to solve this!

Let's save the time increment for our data set in a variable named `dt`, and compute the number of time increments in the data. Then, we'll initialize new arrays of velocity and position to all-zero values, with the intention of updating these to the computed values.

```
In [6]: #time increment
        dt = t[1]-t[0]
        dt

Out[6]: 0.10000000000000001

In [7]: #number of time increments
        N = len(t)
        N

Out[7]: 151

In [8]: #initialize v and x arrays to zero
        v = numpy.zeros(N)
        x = numpy.zeros(N)
```

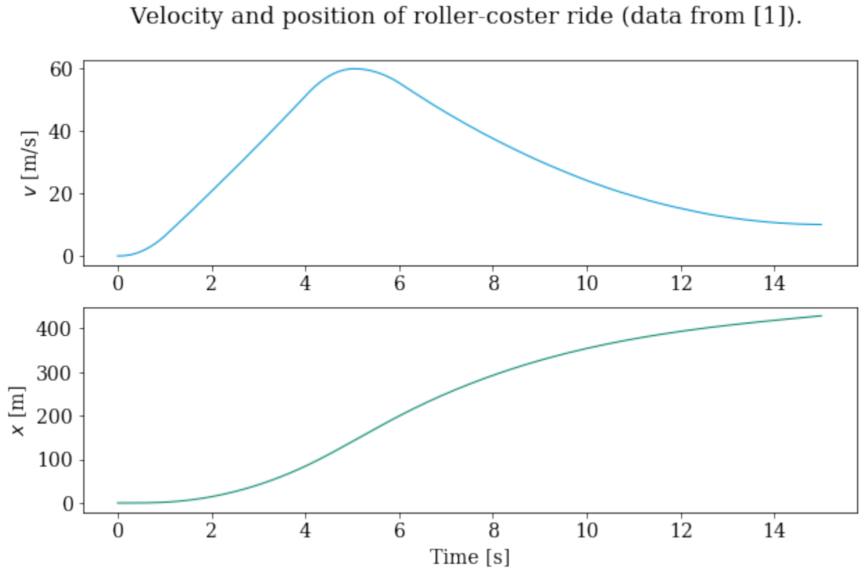
In the code cell below, we use a `for` statement to step through the sequence of acceleration values, each time computing the velocity and position at the subsequent time instant. We are applying the equation for $v(t_i + \Delta t)$ above, and a similar equation for position:

$$x(t_i + \Delta t) \approx x(t_i) + v(t_i)\Delta t \tag{4}$$

```
In [9]: for i in range(N-1):
            v[i+1] = v[i] + a[i]*dt
            x[i+1] = x[i] + v[i]*dt
```

And there you have it. You have computed the velocity and position over time from the acceleration data. We can now make plots of the computed variables. Note that we use the Matplotlib `subplot()` function to get the two plots in one figure. The argument to `subplot()` is a set of three digits, corresponding to the number of rows, number of columns, and plot number in a matrix of sub-plots.

```
In [10]: fig = pyplot.figure(figsize=(10,6))


         pyplot.subplot(211)
         pyplot.plot(t,  v, color='#0096d6', linestyle='-', linewidth=1)
         pyplot.title('Velocity and position of roller-coster ride (data from [1]). \n')
         pyplot.ylabel('$v$ [m/s] ')

         pyplot.subplot(212)
         pyplot.plot(t,  x, color='#008367', linestyle='-', linewidth=1)
         pyplot.xlabel('Time [s]')
         pyplot.ylabel('$x$ [m]');
```



2 Euler's method

The method we used above to compute the velocity and position from acceleration data is known as *Euler's method*. The eminent Swiss mathematician Leonhard Euler presented it in his book *"Institutionum calculi integralis,"* published around 1770 [3].

You can understand why it works by writing out a Taylor expansion for $x(t)$:

$$x(t + \Delta t) = x(t) + \frac{dx}{dt}\Delta t + \frac{d^2x}{dt^2}\frac{\Delta t^2}{2} + \frac{d^3x}{dt^3}\frac{\Delta t^3}{3!} + \cdots \tag{5}$$

With $v = dx/dt$, you can see that the first two terms on the right-hand side correspond to what we used in the code above. That means that Euler's method makes an approximation by throwing away the terms $\frac{d^2x}{dt^2}\frac{\Delta t^2}{2} + \frac{d^3x}{dt^3}\frac{\Delta t^3}{3!} + \cdots$. So the error made in *one step* of Euler's method is proportional to Δt^2. Since we take $N = T/\Delta t$ steps (for a final time instant T), we conclude that the error overall is proportional to Δt.

Euler's method is a first-order method because the error in the approximation goes with the first power of the time increment Δt.

3 Initial-value problems

To get velocity and position from the acceleration data, we needed to know the *initial values* of the velocity and position. Then we could apply Euler's method to *step in time* starting at t_0, with time increment Δt. This setting corresponds to the numerical solution of *initial-value problems*. (We follow here the presentation in [4], p.86.)

Consider the differential equation corresponding to an object in free fall:

$$\ddot{y} = -g, \tag{6}$$

where the dot above a variable represents the time derivative, and g is the acceleration of gravity. Introducing the velocity as intermediary variable, we can write:

$$\begin{aligned} \dot{y} &= v \\ \dot{v} &= -g \end{aligned} \tag{7}$$

The above is a system of two ordinary differential equations, with time as the independent variable. For its numerical solution, we need two initial conditions, and Euler's method:

$$\begin{aligned} y(t_0) = y_0, \qquad y_{i+1} &= y_i + \dot{y}\Delta t \\ v(t_0) = v_0, \qquad v_{i+1} &= v_i + \dot{v}\Delta t \end{aligned} \tag{8}$$

It's so neatly symmetrical that it's just asking for a vectorized equation! Combine the two dependent variables into a vector of unknowns, **y**:

$$\mathbf{y} = \begin{bmatrix} y \\ v \end{bmatrix}, \tag{9}$$

and write the differential equation in vector form, as follows:

$$\dot{\mathbf{y}} = \begin{bmatrix} v \\ -g \end{bmatrix}. \tag{10}$$

Equation (9) above represents the *state* of the system, at any given instant in time. A code design for the numerical solution that generalizes to other changing systems (or *dynamical systems*) is to write one function that computes the right-hand side of the differential equation (the derivatives of the state variables), and another function that takes a state and applies the numerical method for each time increment. The solution is then computed in one `for` statement that calls these functions. Study the code below.

```
In [11]: def freefall(state):
             '''Computes the right-hand side of the freefall differential
             equation, in SI units.
```

```
            Arguments
            ----------
            state : array of two dependent variables [y v]^T

            Returns
            -------
            derivs: array of two derivatives [v -g]^T
            '''

            derivs = numpy.array([state[1], -9.8])
            return derivs
```

```
In [12]: def eulerstep(state, rhs, dt):
            '''Uses Euler's method to update a state to the next one.

            Arguments
            ---------
            state: array of two dependent variables [y v]^T
            rhs  : function that computes the right hand side of the
                   differential equation.
            dt   : float, time increment.

            Returns
            -------
            next_state: array, updated state after one time increment.
            '''

            next_state = state + rhs(state) * dt
            return next_state
```

4 Numerical solution vs. experiment

Here's an idea! Let's use the `freefall()` and `eulerstep()` functions to obtain a numerical solution with the same initial conditions as the falling-ball experiment from Lesson 1, and compare with the experimental data.

You can grab the data from its location online running the following code in a new cell:

```
filename = 'fallingtennisball02.txt'
url = 'http://go.gwu.edu/engcomp3data1'
urlretrieve(url, filename)
```

You already imported `urlretrieve` above. Remember to then comment the assignment of the `filename` variable below. We'll load it from our local copy.

```
In [13]: filename = '../../data/fallingtennisball02.txt'
         t, y = numpy.loadtxt(filename, usecols=[0,1], unpack=True)
```

We'll need to use the same time increment, so let's compute it from two time samples. The initial position is the first value of the `y` array, while the initial velocity is zero. And we'll only look at the section of data before the ball bounces from the ground, which gives us the number of time steps.

```
In [14]: #time increment
         dt = t[1]-t[0]
```

```
In [15]: y0 = y[0] #initial position
         v0 = 0    #initial velocity
         N = 576   #number of steps
```

Now, let's create a new array, called `num_sol`, to hold the results of the numerical solution. The array has dimensions `Nx2`, with each two-element row holding the state variables, (y, v), at a given time instant. After saving the initial conditions in the solution array, we are ready to start stepping in time in a `for` statement. Study the code below.

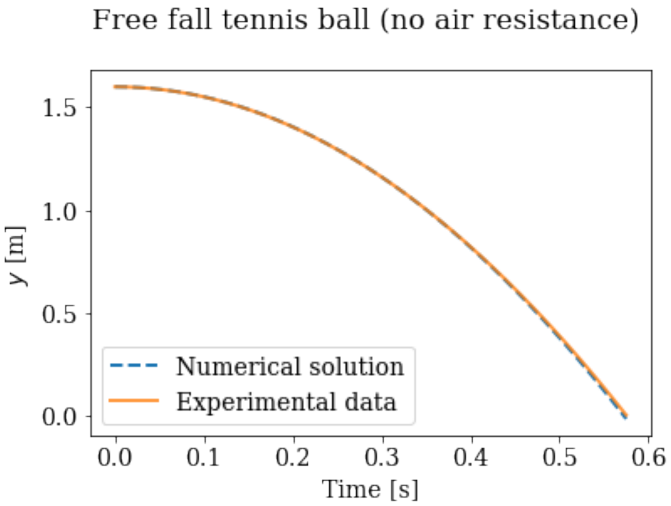
```
In [16]: #initialize array
         num_sol = numpy.zeros([N,2])
```

```
In [17]: #Set intial conditions
         num_sol[0,0] = y0
         num_sol[0,1] = v0
```

```
In [18]: for i in range(N-1):
             num_sol[i+1] = eulerstep(num_sol[i], freefall, dt)
```

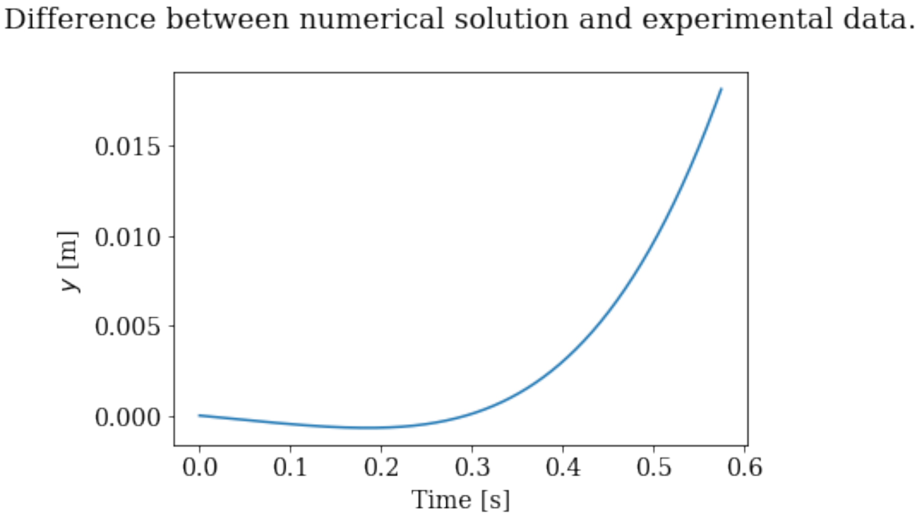
Did it work? Exciting! Let's plot in the same figure both the numerical solution and the experimental data.

```
In [19]: fig = pyplot.figure(figsize=(6,4))
         pyplot.plot(t[:N], num_sol[:,0], linewidth=2, linestyle='--',
               label='Numerical solution')
         pyplot.plot(t[:N], y[:N], linewidth=2, alpha=0.8, label='Experimental data')
         pyplot.xlabel('Time [s]')
         pyplot.ylabel('$y$ [m]')
         pyplot.title('Free fall tennis ball (no air resistance) \n')
         pyplot.legend();
```



The two lines look very close. . . but let's plot the difference to get an idea of the error.

```
In [20]: fig = pyplot.figure(figsize=(6,4))
         pyplot.plot(t[:N], y[:N]-num_sol[:,0])
         pyplot.title('Difference between numerical solution and experimental data.\n')
         pyplot.xlabel('Time [s]')
         pyplot.ylabel('$y$ [m]');
```



5 Air resistance

In Lesson 1 of this module, we computed the acceleration of gravity and got a value less than the theoretical 9.8m/s^2, even when using high-resolution experimental data. Did you figure out why?

We were missing the effect of air resistance! When an object moves in a fluid, like air, it applies a force on the fluid, and consequently the fluid applies an equal and opposite force on the object (Newton's third law).

This force is the *drag* of the fuid, and it opposes the direction of travel. The drag force depends on the object's geometry, and its velocity: for a sphere, its magnitude is given by:

$$F_d = \frac{1}{2}\pi R^2 \rho C_d v^2, \tag{11}$$

where R is the radius of the sphere, ρ the density of the fluid, C_d the drag coefficient of a sphere, and v is the velocity.

Since we have another force involved, we'll have to rethink the problem formulation. The state variables are still the same (position and velocity):

$$\mathbf{y} = \begin{bmatrix} y \\ v \end{bmatrix}. \tag{12}$$

But we'll adjust the differential equation to add the effect of air resistance. In vector form, we can write it as follows:

$$\dot{\mathbf{y}} = \begin{bmatrix} v \\ a_y \end{bmatrix}, \tag{13}$$

where a_y now includes the acceleration due to the drag force:

$$a_y = -g + a_{\text{drag}} \tag{14}$$

With $F_{\text{drag}} = m a_{\text{drag}}$:

$$a_{\text{drag}} = \frac{1}{2m}\pi R^2 \rho C_d v^2 \tag{15}$$

Finally, we can write our differential equation as:

$$\dot{\mathbf{y}} = \begin{bmatrix} v \\ -g + a_{\text{drag}} \end{bmatrix}. \tag{16}$$

Let's write a new function for this modified right-hand side of a falling tennis ball with air resistance.

Note: According to the International Tennis Federation, ITF, the diameter of a tennis ball has to be in the range of 6.54–6.86cm, and its mass in the range of 56.0–59.4g. We chose a value in the middle of the range for each quantity.

```
In [21]: def fall_drag(state):
             '''Computes the right-hand side of the differential equation
             for the fall of a ball, with drag, in SI units.

             Arguments
```

```
    ----------
    state : array of two dependent variables [y v]^T

    Returns
    -------
    derivs: array of two derivatives [v (-g+a_drag)]^T
    '''
    R = 0.0661/2 # radius in meters
    m = 0.0577   # mass in kilograms
    rho = 1.22   # air density kg/m^3
    C_d = 0.47   # drag coefficient for a sphere
    pi = numpy.pi

    a_drag = 1/(2*m) * pi * R**2 * rho * C_d * (state[1])**2

    derivs = numpy.array([state[1], -9.8 + a_drag])
    return derivs
```

Assume the same initial conditions as before:

```
In [22]: y0 = y[0] # initial position
         v0 = 0    # initial velocity
         N = 576   # number of steps
```

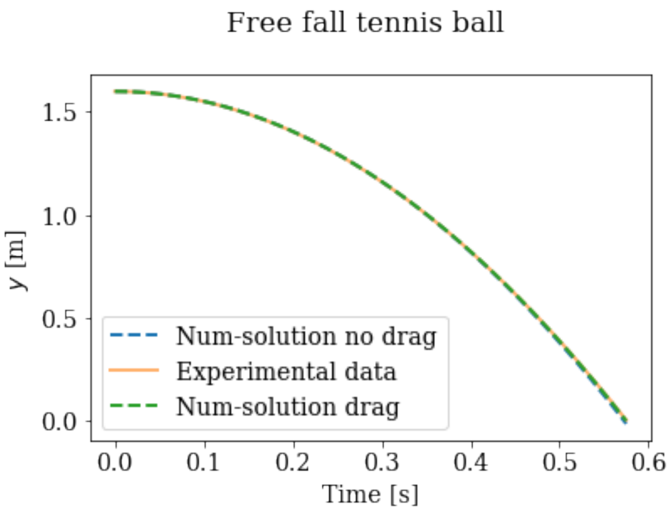
```
In [23]: # initialize array
         num_sol_drag = numpy.zeros([N,2])
```

```
In [24]: # Set intial conditions
         num_sol_drag[0,0] = y0
         num_sol_drag[0,1] = v0
```

```
In [25]: for i in range(N-1):
             num_sol_drag[i+1] = eulerstep(num_sol_drag[i], fall_drag, dt)
```

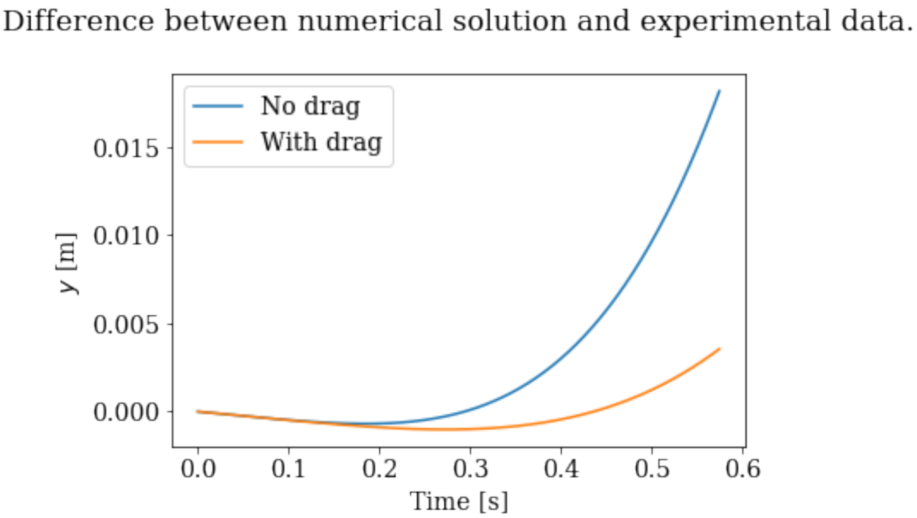
Time to plot and see how it looks! Would you expect the results to be better than in the previous case? Let's plot the three cases and check the differences.

```
In [26]: fig = pyplot.figure(figsize=(6,4))
         pyplot.plot(t[:N], num_sol[:,0], linewidth=2, linestyle='--',
               label='Num-solution no drag')
         pyplot.plot(t[:N], y[:N], linewidth=2, alpha=0.6, label='Experimental data')
         pyplot.plot(t[:N], num_sol_drag[:,0], linewidth=2, linestyle='--',
               label='Num-solution drag')

         pyplot.title('Free fall tennis ball \n')

         pyplot.xlabel('Time [s]')
         pyplot.ylabel('$y$ [m]')
         pyplot.legend();
```



All the lines look very close… but let's plot the differences with the experimental data in both cases, to get an idea of the error.

```
In [27]: fig = pyplot.figure(figsize=(6,4))
         pyplot.plot(t[:N], y[:N]-num_sol[:,0], label='No drag')
         pyplot.plot(t[:N], y[:N]-num_sol_drag[:,0], label='With drag')
         pyplot.title('Difference between numerical solution and experimental data.\n')
         pyplot.xlabel('Time [s]')
         pyplot.ylabel('$y$ [m]')
         pyplot.legend();
```



Discuss with your neighbor

- What do you see in the plot of the difference between the numerical solution and the experimental data?

6 What we've learned

- Integrating an equation of motion numerically.
- Drawing multiple plots in one figure,
- Solving initial-value problems numerically
- Using Euler's method.
- Euler's method is a first-order method.
- Freefall with air resistance is a more realistic model.

7 References

1. *Elementary Mechanics Using Python* (2015), Anders Malthe-Sorenssen, Undergraduate Lecture Notes in Physics, Springer. Data at http://folk.uio.no/malthe/mechbook/

2. *The Physics Hyptertextbook* (n/a), Glenn Elert, Acceleration

3. Euler method. (2017, October 13). In Wikipedia, The Free Encyclopedia. Retrieved 01:21, November 10, 2017, from https://en.wikipedia.org/w/index.php?title=Euler_method&oldid=805120184

4. *Computational Physics with Python*, lecture notes by Eric Ayars, California State University, Chico. Available online on the author's website: https://physics.csuchico.edu/ayars/312/handouts/comp-phys-python.pdf

Lesson 3: Get with the oscillations

So far, in this module of our course in *Engineering Computations* you have learned to:

- capture time histories of a body's position from images and video;
- compute velocity and acceleration of a body, from known positions over time—i.e., take numerical derivatives;
- find the motion description (position versus time) from acceleration data, stepping in time with Euler's method;
- form the state vector and the vectorized form of a second-order dynamical system;
- improve the simple free-fall model by adding air resistance.

You also learned that Euler's method is a *first-order* method: a Taylor series expansion shows that stepping in time with Euler makes an error—called the *truncation error*—proportional to the time increment, Δt.

In this lesson, you'll work with oscillating systems. Euler's method doesn't do very well with oscillating systems, but we'll show you a clever way to fix this. (The modified method is *still* first order, however. We will also confirm the **order of convergence** by computing the error using different values of Δt.

As always, we will need our best-loved numerical Python libraries, and we'll also re-use the `eulerstep()` function from the previous lesson. So let's get that out of the way.

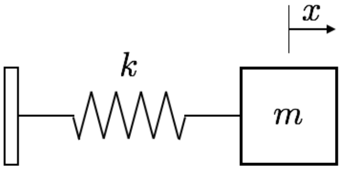
```
In [1]: import numpy
        from matplotlib import pyplot
        %matplotlib inline

        from matplotlib import rcParams
        rcParams['font.family'] = 'serif'
        rcParams['font.size'] = 14
```

```
In [2]: def eulerstep(state, rhs, dt):
            '''Update a state to the next time increment using Euler's method.

            Arguments
            ---------
            state : array of dependent variables
            rhs   : function that computes the RHS of the DiffEq
            dt    : float, time increment

            Returns
            -------
```

```
    next_state : array, updated after one time increment'''

    next_state = state + rhs(state) * dt
    return next_state
```

1 Spring-mass system

A prototypical mechanical system is a mass m attached to a spring, in the simplest case without friction. The elastic constant of the spring, k, determines the restoring force it will apply to the mass when displaced by a distance x. The system then oscillates back and forth around its position of equilibrium.



Simple spring-mass system, without friction.

Newton's law applied to the friction-less spring-mass system is:

$$-kx = m\ddot{x} \tag{1}$$

Introducing the parameter $\omega = \sqrt{k/m}$, the equation of motion is rewriten as:

$$\ddot{x} + \omega^2 x = 0 \tag{2}$$

where a dot above a dependent variable denotes the time derivative. This is a second-order differential equation for the position x, having a known analytical solution that represents *simple harmonic motion*:

$x(t) = x_0 \cos(\omega t)$

The solution represents oscillations with period $P = 2\pi/\omega$ (the time between two peaks), and amplitude x_0.

1.1 System in vector form

It's useful to write a second-order differential equation as a set of two first-order equations: in this case, for position and velocity, respectively:

$$\begin{aligned}\dot{x} &= v \\ \dot{v} &= -\omega^2 x \end{aligned} \tag{3}$$

Like we did in Lesson 2 of this module, we write the state of the system as a two-dimensional vector,

$$\mathbf{x} = \begin{bmatrix} x \\ v \end{bmatrix}, \tag{4}$$

and the differential equation in vector form:

$$\dot{\mathbf{x}} = \begin{bmatrix} v \\ -\omega^2 x \end{bmatrix}. \tag{5}$$

Several advantages come from writing the differential equation in vector form, both theoretical and practical. In the study of dynamical systems, for example, the state vector lives in a state space called the *phase plane*, and many things can be learned from studying solutions to differential equations graphically on a phase plane.

Practically, writing the equation in vector form results in more general, compact code. Let's write a function to obtain the right-hand side of the spring-mass differential equation, in vector form.

```
In [3]: def springmass(state):
            '''Computes the right-hand side of the spring-mass differential
            equation, without friction.

            Arguments
            ---------
            state : array of two dependent variables [x v]^T

            Returns
            -------
            derivs: array of two derivatives [v - ω*ω*x]^T
            '''

            derivs = numpy.array([state[1], -ω**2*state[0]])
            return derivs
```

This worked example follows Reference [1], section 4.3 (note that the source is open access). We set the parameters of the system, choose a time interval equal to 1-20th of the oscillation period, and decide to solve the motion for a duration equal to 3 periods.

```
In [4]: ω = 2
        period = 2*numpy.pi/ω
        dt = period/20  # we choose 20 time intervals per period
        T = 3*period    # solve for 3 periods
        N = round(T/dt)
```

```
In [5]: print(N)
        print(dt)
```

```
60
0.15707963267948966
```

Next, set up the time array and initial conditions, initialize the solution array with zero values, and assign the initial values to the first elements of the solution array.

```
In [6]: t = numpy.linspace(0, T, N)
```

```
In [7]: x0 = 2     # initial position
        v0 = 0     # initial velocity
```

```
In [8]: #initialize solution array
        num_sol = numpy.zeros([N,2])
```

```
In [9]: #Set intial conditions
        num_sol[0,0] = x0
        num_sol[0,1] = v0
```

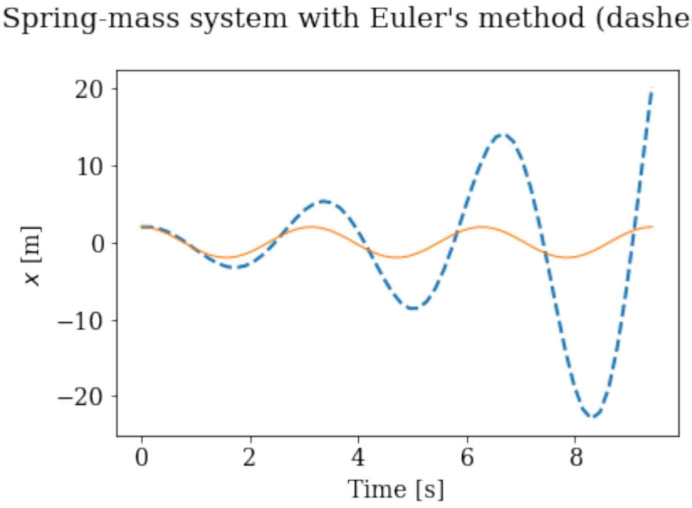
We're ready to solve! Step through the time increments, calling the `eulerstep()` function with the `springmass` right-hand-side derivatives and time increment as inputs.

```
In [10]: for i in range(N-1):
             num_sol[i+1] = eulerstep(num_sol[i], springmass, dt)
```

Now, let's compute the position with respect to time using the known analytical solution, so that we can compare the numerical result with it. Below, we make a plot including both numerical and analytical values in our chosen time range.

```
In [11]: x_an = x0*numpy.cos(ω * t)
```

```
In [12]: fig = pyplot.figure(figsize=(6,4))

         pyplot.plot(t, num_sol[:, 0], linewidth=2, linestyle='--',
                     label='Numerical solution')
         pyplot.plot(t, x_an, linewidth=1, linestyle='-', label='Analytical solution')
         pyplot.xlabel('Time [s]')
         pyplot.ylabel('$x$ [m]')
         pyplot.title('Spring-mass system with Euler\'s method (dashed line).\n');
```



Yikes! The numerical solution exhibits a marked growth in amplitude over time, which certainly is not what the physical system displays. *What is wrong with Euler's method?*

Exercise:

- Try repeating the calculation above using smaller values of the time increment, `dt`, and see if the results improve. Try `dt=P/40`, `P/160` and `P/2000`.
- Although the last case, with 2000 steps per oscillation, does look good enough, see what happens if you then increase the time of simulation, for example to 20 periods. —Run the case again: *What do you see now?*

We consistently observe a growth in amplitude in the numerical solution, worsening over time. The solution does improve when we reduce the time increment `dt` (as it should), but the amplitude still displays unphysical growth for longer simulations.

2 Euler-Cromer method

The thing is, Euler's method has a fundamental problem with oscillatory systems. Look again at the approximation made by Euler's method to get the position at the next time interval:

$$x(t_i + \Delta t) \approx x(t_i) + v(t_i)\Delta t \tag{6}$$

It uses the velocity value at the *beginning* of the time interval to step the solution to the future.

A graphical explanation can help here. Remember that the derivative of a function corresponds to the slope of the tangent at a point. Euler's method approximates the derivative using the slope at the initial point in an interval, and advances the numerical position with that initial velocity. The sketch below illustrates two consecutive Euler steps on a function with high curvature.

Sketch of two Euler steps on a curved function.

Since Euler's method makes a linear approximation to project the solution into the future, assuming the value of the derivative at the start of the interval, it's not very good on oscillatory functions.

A clever idea that improves on Euler's method is to use the updated value of the derivatives for the *second* equation.

Pure Euler's method applies:

$$\begin{aligned} x(t_0) = x_0, \qquad x_{i+1} &= x_i + v_i \Delta t \\ v(t_0) = v_0, \qquad v_{i+1} &= v_i - \omega^2 x_i \Delta t \end{aligned} \tag{7}$$

What if in the equation for v we used the value x_{i+1} that was just computed? Like this:

$$
\begin{aligned}
x(t_0) = x_0, \qquad x_{i+1} &= x_i + v_i \Delta t \\
v(t_0) = v_0, \qquad v_{i+1} &= v_i - \omega^2 x_{i+1} \Delta t
\end{aligned}
\tag{8}
$$

Notice the x_{i+1} on the right-hand side of the second equation: that's the updated value, giving the acceleration at the *end* of the time interval. This modified scheme is called Euler-Cromer method, to honor clever Mr Cromer, who came up with the idea [2].

Let's see what it does. Study the function below carefully—it helps a lot if you write things out on a piece of paper!

```
In [13]: def euler_cromer(state, rhs, dt):
             '''Update a state to the next time increment using Euler-Cromer's method.

             Arguments
             ---------
             state : array of dependent variables
             rhs   : function that computes the RHS of the DiffEq
             dt    : float, time increment

             Returns
             -------
             next_state : array, updated after one time increment'''

             mid_state = state + rhs(state)*dt # Euler step
             mid_derivs = rhs(mid_state)       # updated derivatives

             next_state = numpy.array([mid_state[0], state[1] + mid_derivs[1]*dt])

             return next_state
```

We've copied the whole problem set-up below, to get the solution in one code cell, for easy trial with different parameter choices. Try it out!

```
In [14]: ω = 2
         period = 2*numpy.pi/ω
         dt = period/200  # time intervals per period
         T = 800*period   # simulation time, in number of periods
         N = round(T/dt)

         print('The number of time steps is {}.'.format( N ))
         print('The time increment is {}'.format( dt ))

         # time array
         t = numpy.linspace(0, T, N)

         x0 = 2    # initial position
         v0 = 0    # initial velocity
```

```
#initialize solution array
num_sol = numpy.zeros([N,2])

#Set intial conditions
num_sol[0,0] = x0
num_sol[0,1] = v0

for i in range(N-1):
    num_sol[i+1] = euler_cromer(num_sol[i], springmass, dt)
```

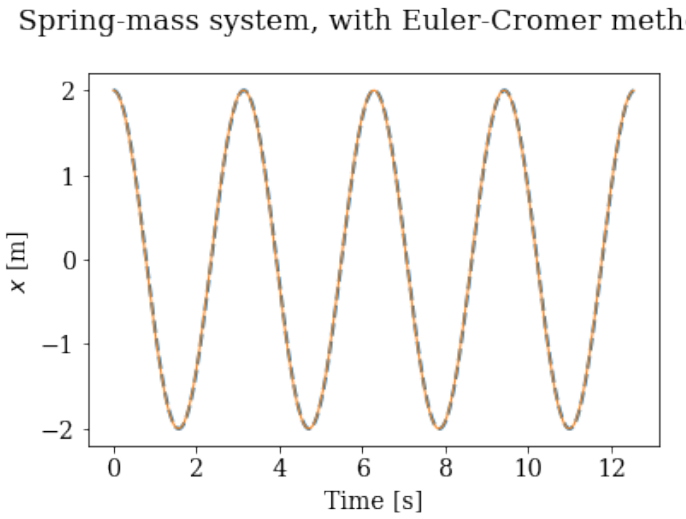
```
The number of time steps is 160000.
The time increment is 0.015707963267948967
```

Recompute the analytical solution, and plot it alongside the numerical one, when you're ready. We computed a crazy number of oscillations, so we'll need to pick carefully the range of time to plot.

First, get the analytical solution. We chose to then plot the first few periods of the oscillatory motion: numerical and analytical.

```
In [15]: x_an = x0*numpy.cos(ω * t) # analytical solution
```

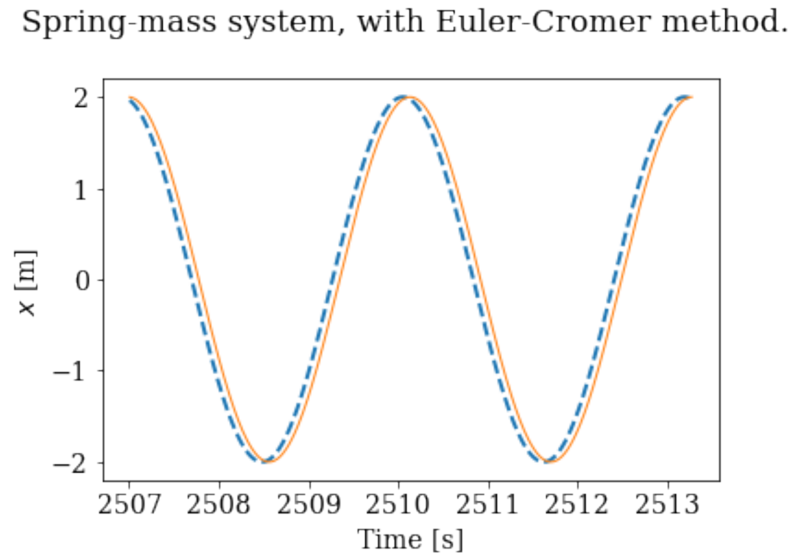
```
In [16]: iend = 800 # in number of time steps

         fig = pyplot.figure(figsize=(6,4))

         pyplot.plot(t[:iend], num_sol[:iend, 0], linewidth=2, linestyle='--',
               label='Numerical solution')
         pyplot.plot(t[:iend], x_an[:iend], linewidth=1, linestyle='-',
               label='Analytical solution')
         pyplot.xlabel('Time [s]')
         pyplot.ylabel('$x$ [m]')
         pyplot.title('Spring-mass system, with Euler-Cromer method.\n');
```



The plot shows that Euler-Cromer does not have the problem of growing amplitudes. We're pretty happy with it in that sense.

But if we plot the end of a long period of simulation, you can see that it does start to deviate from the analytical solution.

```
In [17]: istart = 400

         fig = pyplot.figure(figsize=(6,4))

         pyplot.plot(t[-istart:], num_sol[-istart:, 0], linewidth=2, linestyle='--',
               label='Numerical solution')
         pyplot.plot(t[-istart:], x_an[-istart:], linewidth=1, linestyle='-',
               label='Analytical solution')
         pyplot.xlabel('Time [s]')
         pyplot.ylabel('$x$ [m]')
         pyplot.title('Spring-mass system, with Euler-Cromer method. \n');
```



Looking at the last few oscillations in a very long run shows a slight phase difference, even with a very small time increment. So although the Euler-Cromer method fixes a big problem with Euler's method, it still has some error. It's still a first-order method!

The Euler-Cromer method is first-order accurate, just like Euler's method. The global error is proportional to Δt.

Note: You'll often find the presentation of the Euler-Cromer method with the reverse order of the equations, i.e., the velocity equation solved first, then the position equation solved with the updated value of the velocity. This makes no difference in the results: it's just a convention among physicists.

The Euler-Cromer method is equivalent to a *semi-implicit Euler method*.

3 Convergence

We've said that both Euler's method and the Cromer variant are *first-order accurate*: the error goes as the first power of Δt. In Lesson 2 of this module, we showed this using a Taylor series. Let's now confirm it numerically.

Because simple harmonic motion has a known analytical function that solves the differential equation, we can directly compute a measure of the error made by the numerical solution.

Suppose we ran a numerical solution in the interval from t_0 to $T = N/\Delta t$. We could then compute the error, as follows:

$$e = x_N - x_0 \cos(\omega T) \tag{9}$$

where x_N represents the numerical solution at the N-th time step.

How could we confirm the order of convergence of a numerical method? In the lucky scenario of having an analytical solution to directly compute the error, all we need to do is solve numerically with different values of Δt and see if the error really varies linearly with this parameter.

In the code cell below, we compute the numerical solution with different time increments. We use two nested `for`-statements: one iterates over the values of Δt, and the other iterates over the time steps from the initial condition to the final time. We save the results in a new variable called `num_sol_time`, which is an array of arrays. Check it out!

```
In [18]: dt_values = numpy.array([period/50, period/100, period/200, period/400])
         T = 1*period

         num_sol_time = numpy.empty_like(dt_values, dtype=numpy.ndarray)


         for j, dt in enumerate(dt_values):

             N = int(T/dt)
             t = numpy.linspace(0, T, N)

             #initialize solution array
             num_sol = numpy.zeros([N,2])


             #Set intial conditions
             num_sol[0,0] = x0
             num_sol[0,1] = v0

             for i in range(N-1):
                 num_sol[i+1] = eulerstep(num_sol[i], springmass, dt)

             num_sol_time[j] = num_sol.copy()
```

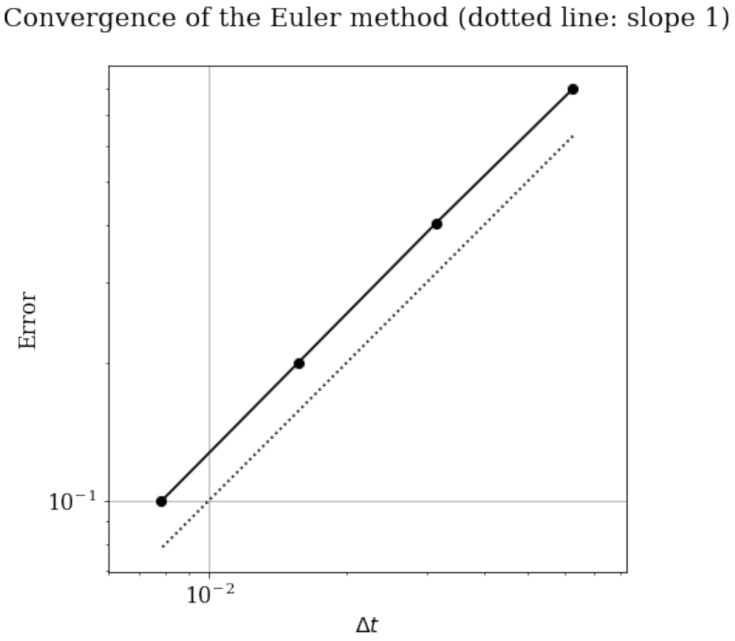
We will need to compute the error with our chosen norm, so let's write a function for that. It includes a line to obtain the values of the analytical solution at the needed instant of time, and then it takes the difference with the numerical solution to compute the error.

```
In [19]: def get_error(num_sol, T):

             x_an = x0 * numpy.cos(ω * T) # analytical solution at final time

             error =  numpy.abs(num_sol[-1,0] - x_an)

             return error
```

All that is left to do is to call the error function with our chosen values of Δt, and plot the results. A logarithmic scale on the plot confirms close to linear scaling between error and time increment.

```
In [20]: error_values = numpy.empty_like(dt_values)

         for j in range(len(dt_values)):

             error_values[j] = get_error(num_sol_time[j], T)
```

```
In [21]: fig = pyplot.figure(figsize=(6,6))

         pyplot.loglog(dt_values, error_values, 'ko-')  #log-log plot
         pyplot.loglog(dt_values, 10*dt_values, 'k:')
         pyplot.grid(True)                         #turn on grid lines
         pyplot.axis('equal')                      #make axes scale equally
         pyplot.xlabel('$\Delta t$')
         pyplot.ylabel('Error')
         pyplot.title('Convergence of the Euler method (dotted line: slope 1)\n');
```



What do you see in the plot of the error as a function of Δt? It looks like a straight line, with a slope close to 1. On a log-log convergence plot, a slope of 1 indicates that we have a first-order method: the error scales as $\mathcal{O}(\Delta t)$—using the "big-O" notation. It means that the error is proportional to the time increment: $e \propto \Delta t$.

4 Modified Euler's method

Another improvement on Euler's method is achieved by stepping the numerical solution to the midpoint of a time interval, computing the derivatives there, and then going back and updating the system state using the midpoint derivatives. This is called *modified Euler's method*.

If we write the vector form of the differential equation as:

$$\dot{\mathbf{x}} = f(\mathbf{x}), \tag{10}$$

then modified Euler's method is:

$$\mathbf{x}_{n+1/2} = \mathbf{x}_n + \frac{\Delta t}{2} f(\mathbf{x}_n) \tag{11}$$

$$\mathbf{x}_{n+1} = \mathbf{x}_n + \Delta t \; f(\mathbf{x}_{n+1/2}). \tag{12}$$

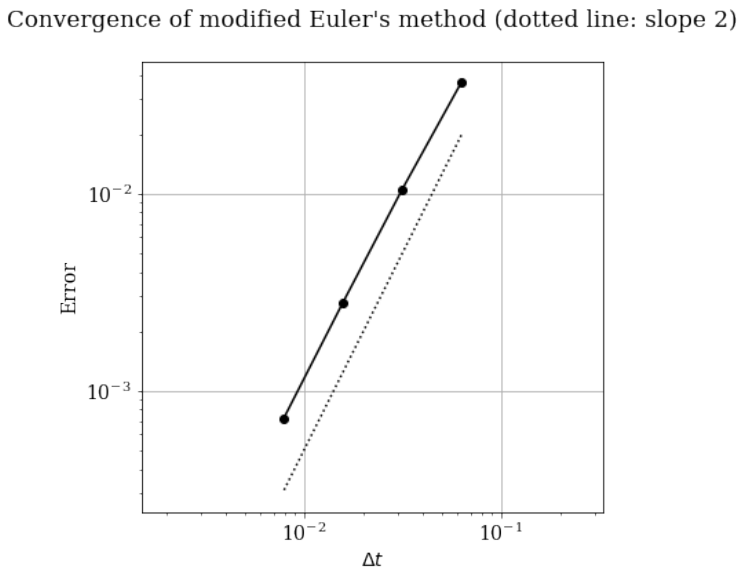
Let's see how it performs with our spring-mass model.

```
In [22]: def rk2_step(state, rhs, dt):
             '''Update a state to the next time increment using modified Euler's method.

             Arguments
             ---------
             state : array of dependent variables
             rhs   : function that computes the RHS of the DiffEq
             dt    : float, time increment

             Returns
             -------
             next_state : array, updated after one time increment'''

             mid_state = state + rhs(state) * dt*0.5
             next_state = state + rhs(mid_state)*dt

             return next_state
```

```
In [23]: dt_values = numpy.array([period/50, period/100, period/200,period/400])
         T = 1*period

         num_sol_time = numpy.empty_like(dt_values, dtype=numpy.ndarray)
         for j, dt in enumerate(dt_values):

             N = int(T/dt)
             t = numpy.linspace(0, T, N)

             #initialize solution array
             num_sol = numpy.zeros([N,2])

             #Set intial conditions
             num_sol[0,0] = x0
             num_sol[0,1] = v0

             for i in range(N-1):
                 num_sol[i+1] = rk2_step(num_sol[i], springmass, dt)

             num_sol_time[j] = num_sol.copy()
```

```
In [24]: error_values = numpy.empty_like(dt_values)

         for j, dt in enumerate(dt_values):

             error_values[j] = get_error(num_sol_time[j], dt)
```

```
In [25]: fig = pyplot.figure(figsize=(6,6))

         pyplot.loglog(dt_values, error_values, 'ko-')
         pyplot.loglog(dt_values, 5*dt_values**2, 'k:')
         pyplot.grid(True)
         pyplot.axis('equal')
         pyplot.xlabel('$\Delta t$')
         pyplot.ylabel('Error')
         pyplot.title('Convergence of modified Euler\'s method (dotted line: slope 2)\n');
```



The convergence plot, in this case, does look close to a slope-2 line. Modified Euler's method is second-order accurate: the effect of computing the derivatives (slope) at the midpoint of the time interval, instead of the starting point, is to increase the accuracy by one order!

Using the derivatives at the midpoint of the time interval is equivalent to using the average of the derivatives at t and $t + \Delta t$: this corresponds to a second-order *Runge-Kutta method*, or RK2, for short. Combining derivatives evaluated at different points in the time interval is the key to Runge-Kutta methods that achieve higher orders of accuracy.

5 What we've learned

- vector form of the spring-mass differential equation
- Euler's method produces unphysical amplitude growth in oscillatory systems
- the Euler-Cromer method fixes the amplitude growth (while still being first order)
- Euler-Cromer does show a phase lag after a long simulation
- a convergence plot confirms the first-order accuracy of Euler's method
- a convergence plot shows that modified Euler's method, using the derivatives evaluated at the midpoint of the time interval, is a second-order method

6 References

1. Linge S., Langtangen H.P. (2016) Solving Ordinary Differential Equations. In: Programming for Computations - Python. Texts in Computational Science and Engineering, vol 15. Springer, Cham, https://doi.org/10.1007/978-3-319-32428-9_4, open access and reusable under CC-BY-NC license.

2. Cromer, A. (1981). Stable solutions using the Euler approximation. *American Journal of Physics*, 49(5), 455-459. https://doi.org/10.1119/1.12478

Lesson 4: Bird’s-eye view of mechanical vibrations

Welcome to **Lesson 4** of the third module in *Engineering Computations*. This course module is dedicated to studying the dynamics of change with computational thinking, Python and Jupyter. The first three lessons give you a solid footing to tackle problems involving motion, velocity, and acceleration. They are:

1. Lesson 1: Catch things in motion
2. Lesson 2: Step to the future
3. Lesson 3: Get with the oscillations

You learned to compute velocity and acceleration from position data, using numerical derivatives, and to capture position data of moving objects from images and video. For physical contexts we used free-fall of a ball, and projectile motion. Then you faced the opposite challenge: computing velocity and position from acceleration data, leading to the idea of stepping forward in time to solve a differential equation.

Our general approach combines these key ideas: (1) turning a second-order differential equation into a system of first-order equations; (2) writing the system in vector form, and the solution in terms of a state vector; (3) designing a code to obtain the solution using separate functions to compute the derivatives of the state vector, and to step the system in time with a chosen scheme (e.g., Euler, Euler-Cromer, Runge-Kutta). It’s a rock-steady approach that will serve you well!

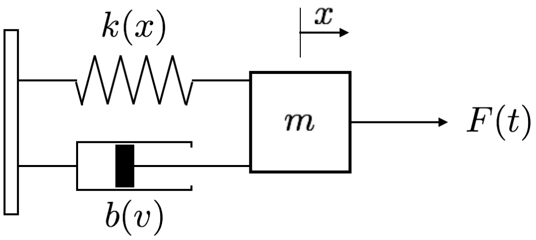
In this lesson, you’ll get a broader view of applying your new-found skills to learn about mechanical vibrations: a classic engineering problem. You’ll study general spring-mass systems with damping and a driving force, and appreciate the diversity of behaviors that arise. We’ll end the lesson presenting a powerful method to study dynamical systems: visualizing direction fields and trajectories in the phase plane.

Are you ready? Start by loading the Python libraries that we know and love.

```
In [1]: import numpy
        from matplotlib import pyplot
        %matplotlib inline

        from matplotlib import rcParams
        rcParams['font.family'] = 'serif'
        rcParams['font.size'] = 14
        rcParams['mathtext.fontset'] = 'cm' # set Computer Modern for LaTeX math
```

1 General spring-mass system

The simplest mechanical oscillating system is a mass m attached to a spring, without friction. We discussed this system in the previous lesson. In general, though, these systems are subject to friction—represented by a mechanical damper—and a driving force. Also, the spring's restoring force could be a nonlinear function of position, $k(x)$.



General spring-mass system, with driving and damping.

Newton's law applied to the general (driven, damped, nonlinear) spring-mass system is:

$$m\ddot{x} = F(t) - b(\dot{x}) - k(x) \tag{1}$$

where

- $F(t)$ is the driving force
- $b(\dot{x})$ is the damping force
- $k(x)$ is the restoring force, possibly nonlinear

Written as a system of two differential equations, we have:

$$\begin{aligned} \dot{x} &= v, \\ \dot{v} &= \frac{1}{m}\left(F(t) - k(x) - b(v)\right). \end{aligned} \tag{2}$$

With the state vector,

$$\mathbf{x} = \begin{bmatrix} x \\ v \end{bmatrix}, \tag{3}$$

the differential equation in vector form is:

$$\dot{\mathbf{x}} = \begin{bmatrix} v \\ \frac{1}{m}\left(F(t) - k(x) - b(v)\right) \end{bmatrix}. \tag{4}$$

In this more general system, the time variable could appear explicitly on the right-hand side, via the driving function $F(t)$. We'll need to adapt the code for the time-stepping function to take the time as an additional argument.

For example, the `euler_cromer()` function we defined in the previous lesson took three arguments: `state, rhs, dt`—the state vector, the Python function computing the right-hand side of the differential equation, and the time step. Let's re-work that function now to take an additional `time` variable, which also gets used in the `rhs` function.

```
In [2]: # new version of the function, taking time as explicit argument
        def euler_cromer(state, rhs, time, dt):
            '''Update a state to the next time increment using Euler-Cromer's method.

            Arguments
            ---------
            state : state vector of dependent variables
            rhs   : function that computes the RHS of the DE, taking (state, time)
            time  : float, time instant
            dt    : float, time step

            Returns
            -------
            next_state : state vector updated after one time increment'''

            mid_state = state + rhs(state, time)*dt # Euler step
            mid_derivs = rhs(mid_state, time)       # update derivatives

            next_state = numpy.array([mid_state[0], state[1] + mid_derivs[1]*dt])

            return next_state
```

2 Case with linear damping

Let's look at the behavior of a system with linear restoring force, linear damping, but no driving force: $k(x) = kx$, $b(v) = bv$, $F(t) = 0$. The differential system is now:

$$\dot{\mathbf{x}} = \begin{bmatrix} v \\ \frac{1}{m}\left(-kx - bv\right) \end{bmatrix}. \tag{5}$$

Now we need to write a function to compute the right-hand side (derivatives) for this system. Even though the system does not explicitly use the time variable in the right-hand side, we still include `time` as an argument to the function, so that it is consistent with our new design for the `euler_cromer()` step. We include `time` in the arguments list, but it is not used inside the function code. It's thus a good idea to specify a *default value* for this argument by writing `time=0` in the arguments list: that will allow us to also call the function leaving the `time` argument blank, if we wanted to (in which case, it will automatically be assigned its default value of 0). Another option for the default value is `time=None`. It doesn't matter because the variable is not used inside the function!

```
In [3]: def dampedspring(state, time=0):
            '''Computes the right-hand side of the spring-mass differential
```

```
    equation, with linear damping.

    Arguments
    ---------
    state : state vector of two dependent variables
    time : float, time instant

    Returns
    -------
    derivs: derivatives of the state vector
    '''

    derivs = numpy.array([state[1], 1/m*(-k*state[0]-b*state[1])])
    return derivs
```

Let's try it! The following example is from section 4.3.9 of Ref. [1] (an open-access text!). We set the model parameters, the initial conditions, and the time-stepping conditions. Then we initialize the numerical solution array `num_sol`, and call the `euler_cromer()` function in the `for` statement. Notice that we pass the time instant `t[i]` to the function's `time` argument (which will allow us to use the same calling signature when we solve for a system with driving force).

```
In [4]: m = 1.0
        k = 1.0
        b = 0.3
```

```
In [5]: x0 = 1    # initial position
        v0 = 0    # initial velocity
```

```
In [6]: T = 12*numpy.pi
        N = 5000
        dt = T/N

        t = numpy.linspace(0, T, N)
```

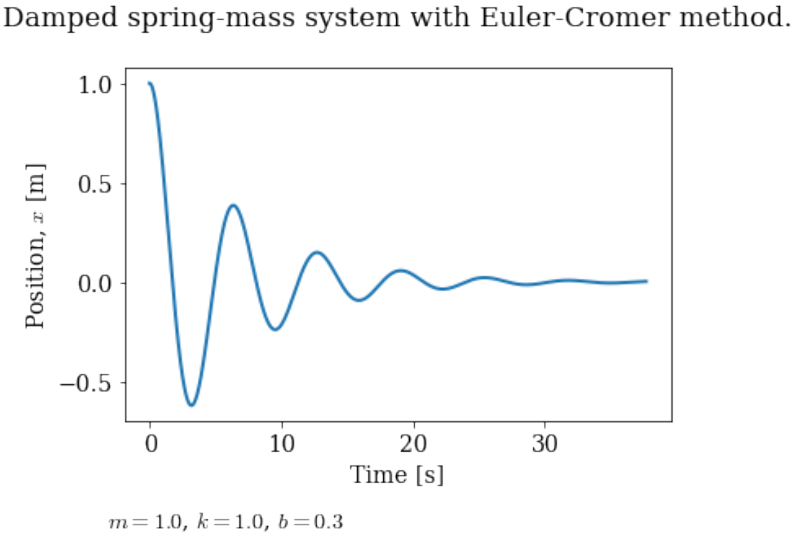
```
In [7]: num_sol = numpy.zeros([N,2]) #initialize solution array

        #Set intial conditions
        num_sol[0,0] = x0
        num_sol[0,1] = v0
```

```
In [8]: for i in range(N-1):
            num_sol[i+1] = euler_cromer(num_sol[i], dampedspring, t[i], dt)
```

Time to plot the solution—in our plot of position versus time below, notice that we added a line with `pyplot.figtext()` at the end. This command adds a custom text to the figure: we use it to print the values of the spring-mass model parameters corresponding to the plot. See how we print the parameter values in the text string? We used Python's string formatter, which you learned about in Module 2 Lesson 1. If we were to re-run the solution with different model parameters, re-executing the code in this cell would update the plot and the text with the proper values. (We don't want to rely on manually changing the text, as that is error prone!)

```
In [9]: fig = pyplot.figure(figsize=(6,4))
```

```
pyplot.plot(t, num_sol[:, 0], linewidth=2, linestyle='-')
pyplot.xlabel('Time [s]')
pyplot.ylabel('Position, $x$ [m]')
pyplot.title('Damped spring-mass system with Euler-Cromer method.\n')
pyplot.figtext(0.1,-0.1,'$m={:.1f}$, $k={:.1f}$, $b={:.1f}$'.format(m,k,b));
```



The result above shows that the oscillations die down over a few periods: the oscillations are *damped* over time. And our plot looks pretty close to Fig. 4.27 of Ref. [1], as it should.

3 Case with sinusoidal driving, and damping

Suppose now that an external force of the form $F(t) = A\sin(\omega t)$ drives the system. This is a typical situation in mechanical systems. Let's find out what a system like that behaves like. The example below comes from section 4.3.10 of Ref. [1].

We're showy, so we decided to use the Unicode character for the Greek letter ω in the code… because we can! With a handy table of Unicode for greek letters, you can pick a symbol code, type it into a code cell, and out comes the symbol. Then, it's a copy-and-paste job to reuse the symbol in the code. And using greek letters for some variable names is very chic.

```
In [10]: u'\u03C9'
```

```
Out[10]: 'ω'
```

```
In [11]: A = 0.5  # parameter values from example in 4.3.10 of Ref. [1]
         ω = 3
```

More than showy, we're snazzy, and so we build a one-line function using the `lambda` keyword. It's just too cool. In Python, you can create a small function in one line using the assignment operator `=`, followed by the `lambda` keyword, then a statement of the form `arguments: expression`—in our case, we have the single argument `time`, and the expression is the sinusoidal driving. The sine mathematical function is avaible to us from the `math` library. Check it out.

```
In [12]: from math import sin
         F = lambda time: A*sin(ω*time)
```

This is really a function: we can call `F()` at any point in our code, passing a value of time, and it will output the result of $F(t) = A\sin(\omega t)$.

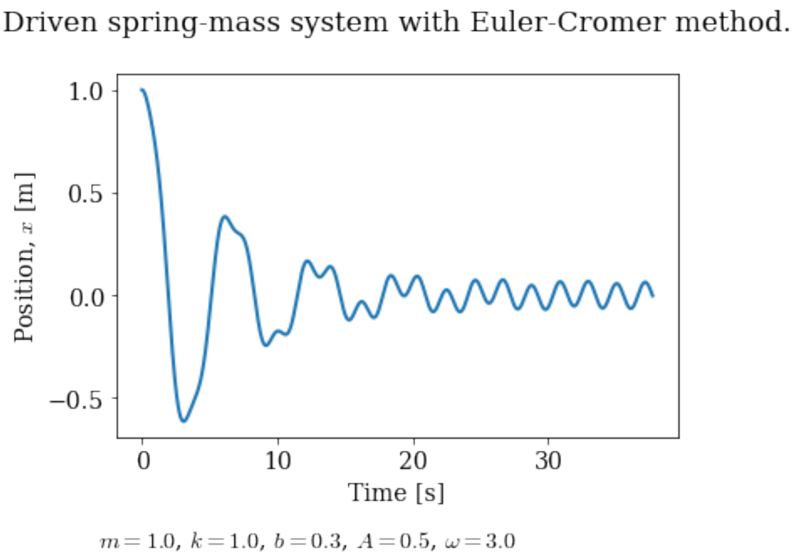
Now, let's write the right-hand side function of derivatives for the driven spring-mass system (with damping). Notice that we use the lambda function `F()` inside this new function, and the `time` variable explicitly as the argument to `F()`. Some powerful Python kung fu!

```
In [13]: def drivenspring(state, time):
             '''Computes the right-hand side of the spring-mass differential
             equation, with sinusoidal driving and linear damping.

             Arguments
             ---------
             state : state vector of two dependent variables
             time : float, time instant

             Returns
             -------
             derivs: derivatives of the state vector
             '''

             derivs = numpy.array([state[1], 1/m*(F(time)-k*state[0]-b*state[1])])
             return derivs
```

Here is where the power of our code design becomes clear: solving the differential equation via time-stepping inside a `for` statement looks just like before, with the only difference being that we pass another right-hand-side function of derivatives. The code cell below solves the driven spring-mass system with the same model parameters we used for the damped system without driving.

```
In [14]: for i in range(N-1):
             num_sol[i+1] = euler_cromer(num_sol[i], drivenspring, t[i], dt)
```

```
In [15]: fig = pyplot.figure(figsize=(6,4))

         pyplot.plot(t, num_sol[:, 0], linewidth=2, linestyle='-')
         pyplot.xlabel('Time [s]')
         pyplot.ylabel('Position, $x$ [m]')
         pyplot.title('Driven spring-mass system with Euler-Cromer method.\n')
         pyplot.figtext(0.1,-0.1,
               '$m={:.1f}$, $k={:.1f}$, $b={:.1f}$, $A={:.1f}$, $\omega={:.1f}$'
               .format(m,k,b,A,ω));
```



Driven spring-mass system with Euler-Cromer method.

$m=1.0$, $k=1.0$, $b=0.3$, $A=0.5$, $\omega=3.0$

And our result looks just like Fig. 4.28 of Ref. [1], as it should. You can see that the system starts out dominated by the spring-mass oscillations, which get damped over time and the effect of the external driving becomes visible, and the sinusoidal driving is all that is left in the end.

Exercise:

- Experiment with different values of the driving-force amplitude, A, and frequency, ω.
- Swap the sine driving for a cosine, and see what happens.

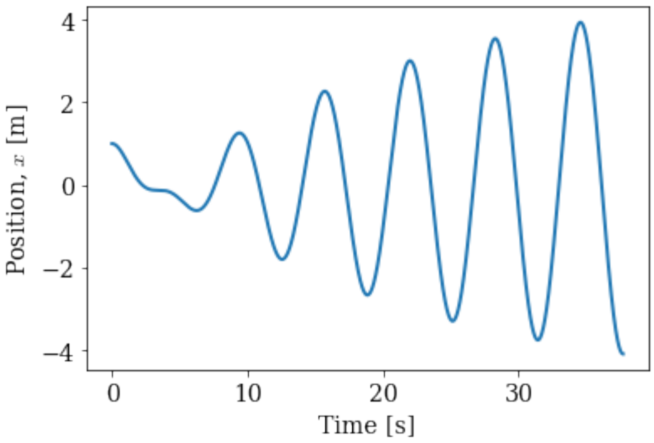
An interesting behavior occurs when the damping is low enough and the frequency of the driving force coincides with the natural frequency of the mass-spring system, $\sqrt{k/m}$: **resonance**.

Try these parameters:

```
In [16]: ω = 1
         b = 0.1
```

```
In [17]: for i in range(N-1):
             num_sol[i+1] = euler_cromer(num_sol[i], drivenspring, t[i], dt)
```

```
In [18]: fig = pyplot.figure(figsize=(6,4))

         pyplot.plot(t, num_sol[:, 0], linewidth=2, linestyle='-')
         pyplot.xlabel('Time [s]')
         pyplot.ylabel('Position, $x$ [m]')
         pyplot.title('Driven spring-mass system with Euler-Cromer method.\n')
         pyplot.figtext(0.1,-0.1,
               '$m={:.1f}$, $k={:.1f}$, $b={:.1f}$, $A={:.1f}$, $\omega={:.1f}$'
               .format(m,k,b,A,ω));
```



Driven spring-mass system with Euler-Cromer method.

$m = 1.0$, $k = 1.0$, $b = 0.1$, $A = 0.5$, $\omega = 1.0$

As you can see, the amplitude of the oscillations grow over time! (Compare the vertical axis of this plot with the previous one.) Our result matches with Fig. 4.29 of Ref. [1].

4 Solutions on the phase plane

The spring-mass system, as you see, can behave in various ways. If the spring is linear, and there is no damping or driving (like in the previous lesson), the motion is periodic. If we add damping, the oscillatory motion decays over time. With driving, the motion can be rather more complicated, and sometimes can exhibit resonance.

Each of these types of motion is represented by corresponding solutions to the differential system, dictated by the model parameters and the initial conditions.

How could we get a sense for all the types of solutions to a differential system? A powerful method to do this is to use the *phase plane*.

A system of two first-order differential equations:

$$\dot{x}(t) = f(x, y) \tag{6}$$

$$\dot{y}(t) = g(x, y) \tag{7}$$

with state vector

$$\mathbf{x} = \begin{bmatrix} x \\ y \end{bmatrix}, \tag{8}$$

is called a *planar autonomous system*: planar, because the state vector has two components; and autonomous (self-generating), because the time variable does not explicitly appear on the right-hand side (which wouldn't apply to the driven spring-mass system).

For initial conditions $\mathbf{x}_0 = (x_0, y_0)$, the system has a unique solution $\mathbf{x}(t) = (x(t), y(t))$. This solution can be represented by a planar curve on the xy-plane—the **phase plane**—and is called a *trajectory* of the system.

On the phase plane, we can plot a **direction (slope) field** by generating a uniform grid of points (x_i, y_j) in some chosen range $(x_{\min}, x_{\max}) \times (y_{\min}, y_{\max})$, and drawing small line segments representing the direction of the vector field $(f(x,y), g(x,y)$ on each point.

Let's draw a direction field for the damped spring-mass system, and include a solution trajectory. We copied the whole problem set-up below, to get a solution all in one code cell, for easy trial with different parameter choices.

```
In [19]: m = 1
         k = 1
         b = 0.3

         x0 = 3     # initial position
         v0 = 3     # initial velocity

         T = 12*numpy.pi
         N = 5000
         dt = T/N

         t = numpy.linspace(0, T, N)
         num_sol = numpy.zeros([N,2]) #initialize solution array

         #Set intial conditions
         num_sol[0,0] = x0
         num_sol[0,1] = v0

         for i in range(N-1):
             num_sol[i+1] = euler_cromer(num_sol[i], dampedspring, t[i], dt)
```

To choose a range for the plotting area of the direction field, let's look at the maximum values of the solution array.

```
In [20]: numpy.max(num_sol[:,0])

Out[20]: 4.0948277569088525

In [21]: numpy.max(num_sol[:,1])

Out[21]: 3.0
```

With that information, we choose the plotting area as $(-4, 4) \times (-4, 4)$. Below, we'll create an array named `coords` to hold the positions of mesh lines on each coordinate direction. Here, we pick 11 mesh points in each direction.

Then, we'll call the very handy `meshgrid()` function of NumPy—you should definitely study the documentation and use pen and paper to diligently figure out what it does!

The outputs of `meshgrid()` are two matrices holding the x and y coordinates, respectively, of points on the grid. Combined, these two matrices give the coordinate pairs of every grid point where we'll compute the direction field.

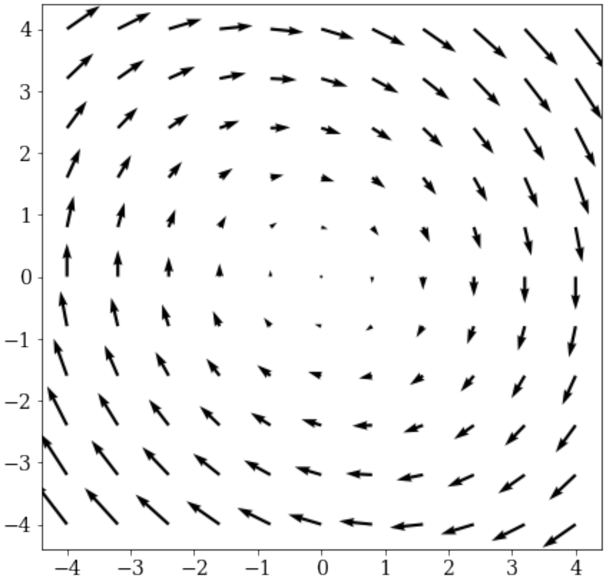
```
In [22]: coords = numpy.linspace(-4,4,11)
         X, Y = numpy.meshgrid(coords, coords)
```

Look at the vector form of the differential system again... with our two matrices of coordinate values for the grid points, we could compute the vector field on all these points in one go using array operations:

```
In [23]: F = Y
         G = 1/m * (-k*X -b*Y)
```

Matplotlib has a type of plot called `quiver` that draws a vector field on a plane. Let's try it out using the vector field we computed above.

```
In [24]: fig = pyplot.figure(figsize=(7,7))

         pyplot.quiver(X,Y, F,G);
```



OK, that's not bad. The arrows on each grid point represent vectors $(f(x,y), g(x,y))$, computed from the right-hand side of the differential equation.

What are the axes on this plot? Well, they are the components of the state vector—which for the spring-mass system are *position* and *velocity*. The vector field looks like a "flow" going around the origin, the values of position and velocity oscillating around. If you imagine an initial condition represented by a coordinate pair (x_0, y_0), the solution trajectory would follow along the arrows, spiraling around the origin, while slowly approaching it.

We'd like to visualize a trajectory on the vector-field plot, and also improve it in a few ways. But before that, Python will astonish you with a splendid fact: you can also compute the vector field on the grid points by calling the function `dampedspring()`, passing as argument a list made of the matrices `X` and `Y`.

Why does this work? Study the function and think!

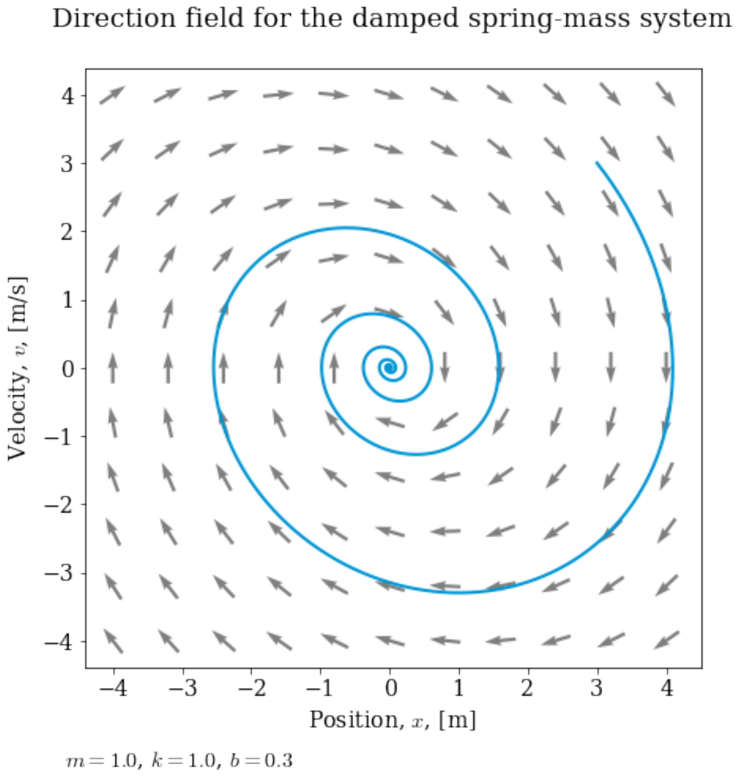
```
In [25]: F, G = dampedspring([X,Y])
```

The default behavior of `quiver` is to scale the vectors (arrows) with the magnitude, but direction fields are usually drawn using line segments of equal length. Also by default, the vectors are drawn *starting at* the grid points, while direction fields ususally *center* the line segments. We can improve our plot using by *scaling* the vectors by their magnitude, and using the `pivot='mid'` option on the plot. A little transparency is also nice.

To plot the improved direction field below, we drew ideas from a tutorial available online, see Ref. [2]. To compute the magnitude of the vectors, we use the `numpy.hypot()` function, which returns the triangle hypotenuse of two right-angled sides.

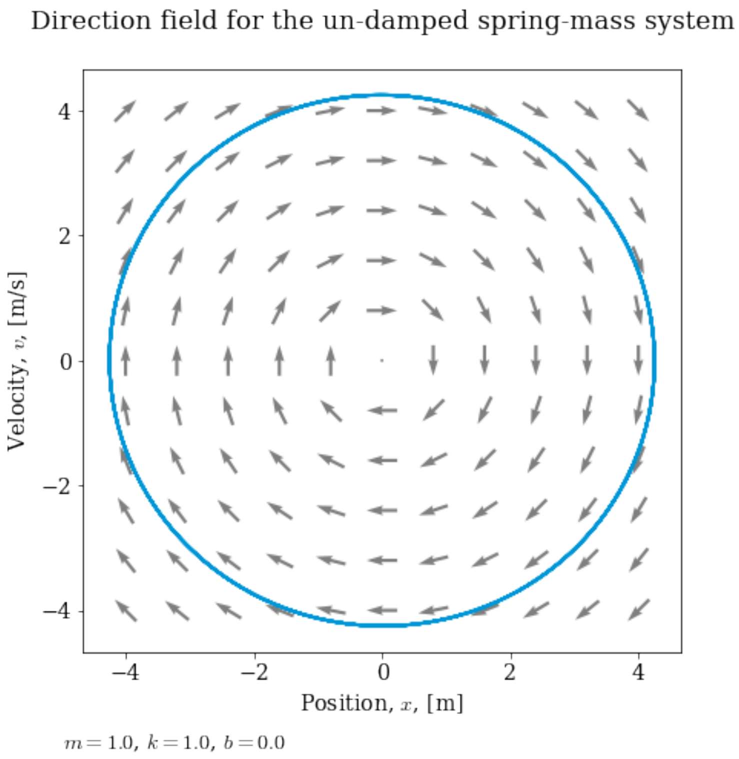
We should also add axis labels and a title!

```
In [26]: M = numpy.hypot(F,G)
         M[ M == 0] = 1       # to avoid zero-division
         F = F/M
         G = G/M

         fig = pyplot.figure(figsize=(7,7))

         pyplot.quiver(X,Y, F,G, pivot='mid', alpha=0.5)
         pyplot.plot(num_sol[:,0], num_sol[:,1], color= '#0096d6', linewidth=2)
         pyplot.xlabel('Position, $x$, [m]')
         pyplot.ylabel('Velocity, $v$, [m/s]')
         pyplot.title('Direction field for the damped spring-mass system\n')
         pyplot.figtext(0.1,0,'$m={:.1f}$, $k={:.1f}$, $b={:.1f}$'.format(m,k,b));
```



And just for kicks, let's re-do everything with zero damping:

```
In [27]: m = 1
         k = 1
         b = 0

         x0 = 3    # initial position
         v0 = 3    # initial velocity

         T = 12*numpy.pi
         N = 5000
         dt = T/N

         t = numpy.linspace(0, T, N)
         num_sol = numpy.zeros([N,2]) #initialize solution array

         #Set intial conditions
         num_sol[0,0] = x0
         num_sol[0,1] = v0

         for i in range(N-1):
             num_sol[i+1] = euler_cromer(num_sol[i], dampedspring, t[i], dt)

         F, G = dampedspring([X,Y])

         M = numpy.hypot(F,G)
         M[ M == 0] = 1      # to avoid zero-division
         F = F/M
         G = G/M

         fig = pyplot.figure(figsize=(7,7))

         pyplot.quiver(X,Y, F,G, pivot='mid', alpha=0.5)
         pyplot.plot(num_sol[:,0], num_sol[:,1], color= '#0096d6', linewidth=2)
         pyplot.xlabel('Position, $x$, [m]')
         pyplot.ylabel('Velocity, $v$, [m/s]')
         pyplot.title('Direction field for the un-damped spring-mass system\n')
         pyplot.figtext(0.1,0,'$m={:.1f}$, $k={:.1f}$, $b={:.1f}$'.format(m,k,b));
```



Challenge task

- Write a function to draw direction fields as above, taking as arguments the right-hand-side derivatives function, and lists containing the plot limits and the number of grid lines in each coordinate direction.

- Write some code to capture mouse clicks on the direction field, following what you learned in Lesson 1 of this module.

- Use the captured mouse clicks as initial conditions and obtain the corresponding trajectories by solving the differential system, then make a new plot showing the trajectories with different colors.

5 What we've learned

- General spring-mass systems have several behaviors: periodic in the undamped case, decaying oscillations when damped, complex oscillations when driven.
- Resonance appears when the driving frequency matches the natural frequency of the system.
- We can add formatted strings in figure titles, labels and added text.
- The `lambda` keyword builds one-line Python functions.
- The `meshgrid()` function of NumPy is handy for building a grid of points on a plane.
- State vectors of a differential system live on the *phase plane*.
- Solutions of the differential system (given initial conditions) are *trajectories* on the phase plane.

- Trajectories for the undamped spring-mass system are circles; in the damped case, they are spirals toward the origin.

6 References

1. Linge S., Langtangen H.P. (2016) Solving Ordinary Differential Equations. In: Programming for Computations - Python. Texts in Computational Science and Engineering, vol 15. Springer, Cham, https://doi.org/10.1007/978-3-319-32428-9_4, open access and reusable under CC-BY-NC license. V
2. Plotting direction fields and trajectories in the phase plane, as part of the Lotka-Volterra tutorial by Pauli Virtanen and Bhupendra, in the *SciPy Cookbook*.