ISTA 421 / INFO 521 - Homework 1

Charles Kenneth "Ken" Youens-Clark

Graduate

1. [0 points] If you haven't already, setup your programming environment! Python is highly recommended: `http://w3.sista.arizona.edu/~clayton/courses/ml/python_setup.html`

   If python is new to you, work through the short python tutorial: `http://w3.sista.arizona.edu/~clayton/courses/ml/tutorial.html`

   See the Appendix at the end of these instructions for additional information about numpy, arrays and matrices.

2. [1 point] **Exercise 1.1** from FCMA p.35

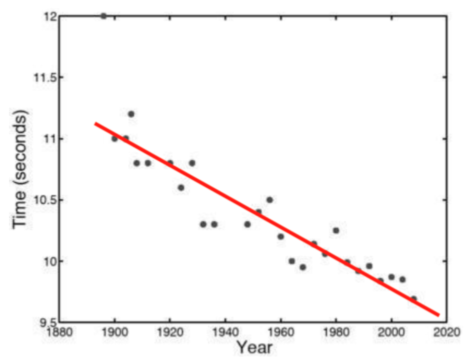

Figure 1: Reproduction of figure 1.1, Olympic men's 100m data

By examining Figure 1.1 [from p. 2 of FCMA, reproduced here], estimate (by hand / in your head) the kind of values we should expect for w_0 (y-intercept) and w_1 (slope) as parameters of a line fit to the data (e.g., High? Low? Positive? Negative?). (No computer or calculator calculation is needed here – just estimate!)

Solution.

I added a red line to the image above to indicate the kind of slope I would expect. Y intercept would be around 11.25, slope is ?.

NOTE: The following three exercises (3, 4 and 5) review basic linear algebra concepts.

Notation conventions:

- Script variables, such as x_{n2} and w_1 represent scalar values
- Lowercase bold-face variables, such as $\mathbf{x}_n$ and $\mathbf{w}$, represent vectors
- Uppercase bold-face variables, such as $\mathbf{X}$, represent n (rows) $\times$ m (columns) matrices
- Note that because all indexes in the following are either a single digit integers (0, 1, ..., 9), or a single letter representing an integer index, e.g., n, I am representing multiple dimension indexes without a comma, as it is unambiguous; e.g., x_{32} is the element scalar value of $\mathbf{X}$ at row 3, column 2. When we have to refer to specific index values greater than 9, we'll use commas, such as $x_{32,3}$ is the scalar value in the 32nd row and 3rd column.
- '$\top$' in expressions like $\mathbf{w}^\top$ indicates the *transpose* operator.
- Unless stated otherwise, we will assume that all vectors *without* the transpose, $\top$, are *column* vectors, so $\mathbf{w}^\top$ is a *row vector.*
- It is sometimes convenient to express an example vector as a bracketed list of elements (e.g., in a sentence): $[x_1, x_2, ..., x_n]$. In general I am going to try to be careful about the orientation of vectors, so the previous example would be a *row* vector. To make it a column vector, I'll add a transpose: $[x_1, x_2, ..., x_n]^\top$.

3. [2 points] **Exercise 1.3** from FCMA p.35

   Show that:

$$\mathbf{w}^\top \mathbf{X}^\top \mathbf{X}\mathbf{w} = w_0^2 \left(\sum_{n=1}^{N} x_{n1}^2 \right) + 2w_0 w_1 \left(\sum_{n=1}^{N} x_{n1} x_{n2} \right) + w_1^2 \left(\sum_{n=1}^{N} x_{n2}^2 \right),$$

   where

$$\mathbf{w} = \begin{bmatrix} w_0 \\ w_1 \end{bmatrix}, \mathbf{X} = \begin{bmatrix} x_{11} & x_{12} \\ x_{21} & x_{22} \\ x_{31} & x_{32} \\ \vdots & \vdots \\ x_{N1} & x_{N2} \end{bmatrix}.$$

   (Hint – it's probably easiest to do the $\mathbf{X}^\top \mathbf{X}$ first!)

   Solution. <Solution goes here>

4. [1 point] **Exercise 1.4** from FCMA p.35

   Using $\mathbf{w}$ and $\mathbf{X}$ as defined in the previous exercise, show that $(\mathbf{X}\mathbf{w})^\top = \mathbf{w}^\top \mathbf{X}^\top$ by multiplying out both sides.

   Solution. <Solution goes here>

5. [2 points] **Exercise 1.5** from FCMA p.35

When multiplying a scalar by a vector (or matrix), we multiply each element of the vector (or matrix) by that scalar. For $\mathbf{x}_n = [x_{n1}, x_{n2}]^\top$, $\mathbf{t} = [t_1, ..., t_N]^\top$, $\mathbf{w} = [w_0, w_1]^\top$, and

$$\mathbf{X} = \begin{bmatrix} \mathbf{x}_1{}^\top \\ \mathbf{x}_2{}^\top \\ \vdots \\ \mathbf{x}_N{}^\top \end{bmatrix}$$

show that

$$\sum_n \mathbf{x}_n t_n = \mathbf{X}^\top \mathbf{t}$$

and

$$\sum_n \mathbf{x}_n \mathbf{x}_n^\top \mathbf{w} = \mathbf{X}^\top \mathbf{X} \mathbf{w}$$

Solution. <Solution goes here>

6. [5 points] Reading and Displaying Numpy Arrays:

   Write a script, called 'hw1.py', that uses the numpy function `loadtxt` to load the contents of `humu.txt` (available in the `data/` subdirectory of the release) into a variable ($).

   Hint: If `loadtxt` *is new to you, visit*
   `http://docs.scipy.org/doc/numpy-1.13.0/reference/generated/numpy.loadtxt.html` *;*
   in general python, numpy, scipy and matplotlib all have great online documentation! Also, keep in mind that you will need to `import numpy` *at the top of your script in order to access numpy functions; I recommend avoiding using* `from numpy import *` *as that pollutes your namespace and can lead to very hard to debug errors.*

   The default output of `loadtxt` is a numpy ndarray. You can show this by getting the `type` of the variable. From the python terminal you could do this by entering `type(var)`, or if you are executing your script and want to see the output, add a print statement: `print(type(var))`. In python 3.6, the output should be `<class 'numpy.ndarray'>` . Have your script print the type ($).

   You can get the total size of a numpy object using the `size` field: `var.size`. `size` holds the total number of elements in the array. Google (or other search) "numpy size" to see the documentation about this field. Have your script print the size of the loaded humu data ($).

   Knowing the (total) size is helpful, but in this case we are working with a 2-dimensional array, so it would be useful to get the array dimensions. The numpy ndarray field `shape` holds a tuple representing size of each dimension of the array. Look at the documentation for numpy `shape`. Have your script print the shape of the loaded humu data ($).

   Hint: To store both the width and height into variables in one go, try: `h, w = var.shape`*. This is an example of "unpacking", and you can even do this with nested patterns:* `a, (b, c) = [1, [2, 3]]`*. You could alternately do:* `h = var.shape[0]` *and* `w = var.shape[1]`*.*

   What is the range of values contained within the array? Use the `min` and `max` functions to figure this out. Have your script print the min and max values of the array ($).

   Hint: take a look at the related `amin` *and* `amax` *functions for full documentation.*

   Use the `min` and `max` values to scale the values in your humu array so that they lie in the range $[0, 1]$ – store this in a new array ($). Verify that the new, scaled array has the same dimensions as the original (print the shape!) but that the new values are in the range $[0, 1]$ (print the min and max of the new array!) ($).

   You will now use matplotlib to plot the original (not the scaled) image. To do so, you need to do the following:

   (a) Add `import matplotlib.pyplot as plt` to the top of your file.

   (b) Add `plt.figure()` to create a new figure object

   (c) Use `imshow` to display your original humu array.

   If you successfully do the above and run your script, it will likely open the figure very quickly and then close it right after (if it shows anything at all). To get the figure to hang around, add: `plt.show()`. Note: the figure will stay open and python will not continue further script execution until after you close the figure.

   Why does the plot look so strange? Print the current colormap: plt.cm.cmapname – this will print the default colormap name (which is likley 'afmhot', matplotlib's version of Matlab's `jet`, or 'Vega20c_r'). This is the default colormap that is useful for data visualization. Try plotting your figure again, but this time, select the 'gray' scale colormap by adding the following argument to `imshow`: `cmap='gray'` ($).

Note: specifying the colormap in imshow does not change the default colormap.

Tip: visit `http://matplotlib.org/1.2.1/examples/pylab_examples/show_colormaps.html` *to see a list of available colormaps. You can also create your own colormaps:*
`http://matplotlib.org/examples/pylab_examples/custom_cmap.html`
`http://stackoverflow.com/questions/16834861/create-own-colormap-using-matplotlib-and-plot-color-sc`

Now make a grayscale image that is the same size as the one we have been using, except create the values to be uniformly random (see the function `numpy.random.random()`). Does it look like what you expected? Create a second such image. Does it look much different? ($)

Write the uniformly random array to a file called 'random.txt' using the numpy `savetxt` function. Confirm (1) that you can read the result by opening the file in a text editor, and (2) that you can load it back into another variable and display it to get the same image as before ($).

(For fun: Anyone know what 'humu' is? Give the full name!)

Solution.

```
import numpy as np
import matplotlib.pyplot as plt

dat = np.loadtxt("../data/humu.txt")
print('type = {}'.format(type(dat)))
print('size = {}'.format(dat.size))
print('shape = {}'.format(dat.shape))
print('max = {}'.format(dat.max()))  # also np.amax(dat)
print('min = {}'.format(dat.min()))  # also np.amin(dat)

scaled = dat / dat.max()
print('scaled min = {} max = {} shape = {}'.format(scaled.min(),
                                                   scaled.max(),
                                                   scaled.shape))

plt.figure()
plt.imshow(dat)
plt.show()

print(plt.cm.cmapname)

plt.imshow(dat, cmap='gray')
plt.show()

outfile = 'random.png'
for _ in range(0, 2):
    ran = np.random.random(dat.shape)
    plt.imshow(ran)
    plt.show()
    np.savetxt(outfile, ran)

ran1 = np.loadtxt(outfile)
plt.imshow(ran1)
plt.show()
```

```
print('Done.')

$ ./hw1.1.py
type = <class 'numpy.ndarray'>
size = 210816
shape = (366, 576)
max = 0.9450980392156862
min = 0.0
scaled min = 0.0 max = 1.0 shape = (366, 576)
tab20c_r
Done.
```

The humuhumunukunukuapua'a is the reef trigger fish and the state fish of Hawai'i.

PART B

In this last part of the exercise you will load a 1-dimensional array of float values in `walk.txt` (of the `data/` subdirectory of the release) representing a bounded random walk, where bounded means the walk does not venture outside of some range of values. What is the range of values contained within the array? Use the `min` and `max` functions to figure this out. Have your script print the min and max values of the array ($).

Hint: take a look at the related `amin` *and* `amax` *functions for full documentation.*

Use the `min` and `max` values to *linearly* scale the values of the `walk.txt` array so that they lie in the range $[0, 1]$ – store this in a new array and save the result as `walk_scale01.txt` – you will include this file in our submission ($). Verify that the new, scaled array has the same dimensions as the original (print the shape!) but that the new values are in the range $[0, 1]$ (print the min and max of the new array!)

Also plot the original `walk.txt` and your scaled version `walk_scale01.txt`. To plot you can use the following very simple function that uses the matplotlib plot function:

```
import matplotlib.pyplot as plt

def plot_1d_array(arr):
    plt.figure()
    plt.plot(arr)
```

Include these two plots, with appropriate captions, in your results ($).

Note: the shapes of the original `walk.txt` *and* `walk_scale01.txt` *should be identical, but the y-axis ranges should reflect the original and scaled ranges.*

Solution.

```
#!/usr/bin/env python3

"""
walk.py - manipulate and plot "walk.txt"
Ken Youens-Clark
27 August 2018
"""

import numpy as np
import matplotlib.pyplot as plt
```

```
# --------------------------------------------------
def main():
    """
    main()
    """
    dat = np.loadtxt('../data/walk.txt')
    print('Data  : min "{:5}, max "{:5}", shape "{}"'.format(
        dat.min(), dat.max(), dat.shape))

    abs_min = np.abs(dat.min())
    scaled = (dat + abs_min) / (dat.max() + abs_min)
    print('Scaled: min "{:5}, max "{:5}", shape "{}"'.format(
        scaled.min(), scaled.max(), scaled.shape))

    outfile = '../data/walk_scale01.txt'
    np.savetxt(outfile, scaled)
    print('Scaled data saved to "{}"'.format(outfile))

    plot_1d_array(arr=dat, title='Original', outfile='walk.png')
    plot_1d_array(arr=scaled, title='Scaled', outfile='walk_scaled.png')


# --------------------------------------------------
def plot_1d_array(arr, title=None, outfile=None):
    """
    Plot a 1D array
    :param: arr - a 1D Numpy array
    :param: title - figure title (str)
    :param: outfile - path to write image (str)

    :return: void
    """

    plt.figure()
    if title:
        plt.title(title)
    plt.plot(arr)

    if outfile:
        plt.savefig(outfile)
        print('Wrote to "{}"'.format(outfile))

    plt.show()


# --------------------------------------------------
if __name__ == '__main__':
    main()
```

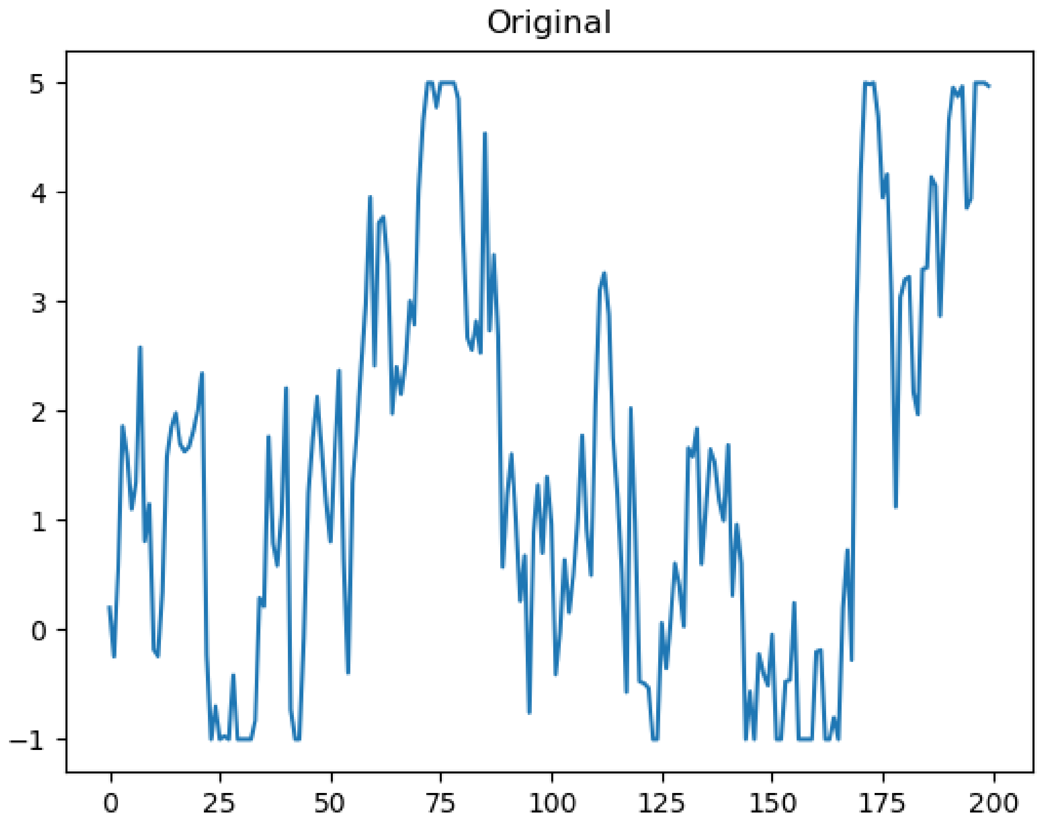

Figure 2: Plot of original walk data

```
$ ./walk.py
Data  : min " -1.0, max "  5.0", shape "(200,)"
Scaled: min "  0.0, max "  1.0", shape "(200,)"
Scaled data saved to "../data/walk_scale01.txt"
Wrote to "walk.png"
Wrote to "walk_scaled.png"
```

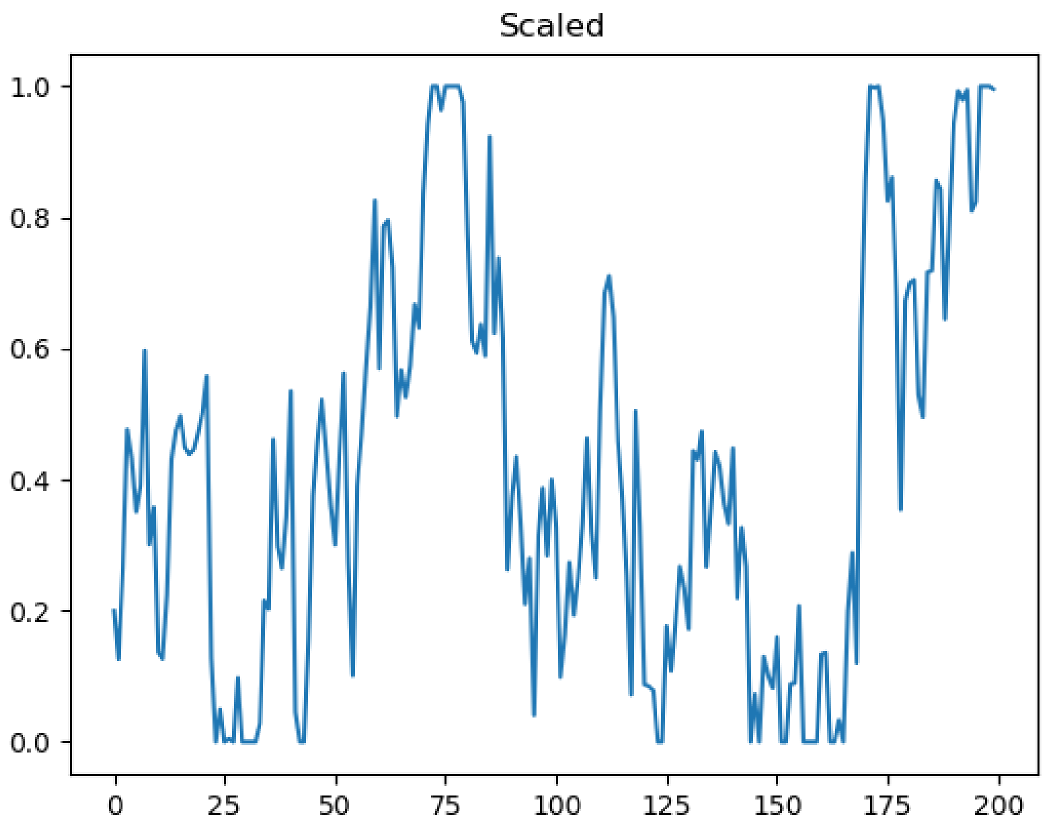

Figure 3: Plot of scaled walk

7. [1 point] Functions:

   You can define new python functions with:

```
def <function_name>(<argument1>, <argument2>, ...):
    <function_body>...
```

   Modify your script from problem 6 so that all of the code between loading the 'humu.txt' to saving and reading the random 'out.txt' file are within a single function called `exercise6` and takes two arguments, `infile` and `outfile`, which are used instead of the hard-coded filenames. If you execute your script now, nothing will appear to happen – in fact, python does execute and defines the function, but the function is not called. Add the line `exercise6('humu.txt', 'out.txt')` to the end of the script, and now the function will be called when you execute the script ($).

   Tip: Python functions can also have **optional** *arguments; for example, we could have specified the exercise6 argument list as:* `exercise6(infile='humu.txt', outfile='out.txt')`

```
def main():
    """
    main
    :param: none
    :return: void
    """
    exercise6("../data/humu.txt", "out.txt")

def scale01(arr):
    """
    Linearly scale the values of an array in the range [0,1]
    :param arr: input ndarray
    :return: scaled ndarray
    """
    return arr / arr.max()

def exercise6(infile, outfile):
    """
    Read a file into a Numpy ndarray
    :param infile: the file to read
    :param outfile: where to write the output
    :return: void
    """
    dat = np.loadtxt(infile)

    scaled = scale01(dat)
    print('scaled min = {} max = {} shape = {}'.format(scaled.min(),
                                                         scaled.max(),
                                                         scaled.shape))

    plt.figure()
    plt.imshow(dat)
    plt.show()

    print(plt.cm.cmapname)
```

```
    plt.imshow(dat, cmap='gray')
    plt.show()

    for _ in range(0, 2):
        ran = np.random.random(dat.shape)
        plt.imshow(ran)
        plt.show()
        np.savetxt(outfile, ran)

    ran1 = np.loadtxt(outfile)
    plt.imshow(ran1)
    plt.show()

    print('Done.')

if __name__ == '__main__':
    main()
```

```
$ ./hw1.py
type = <class 'numpy.ndarray'>
size = 210816
shape = (366, 576)
max = 0.9450980392156862
min = 0.0
scaled min = 0.0 max = 1.0 shape = (366, 576)
tab20c_r
Done.
```

8. [2 points] Documenting:

   You must always document your code! Documenting your code is required for this class – you will loose points if you do not document your code. Python in-line comments can be added using the # character – anything following will be ignored by python. Python also uses a special idiom for documenting functions: Right after function signature line add documentation within a triple-quote body, e.g.,:

```
def scale01(arr):
    """
    Linearly scale the values of an array in the range [0,1]
    :param a: input ndarray
    :return: scaled ndarray
    """
    <function_body>...
```

   Beyond making your source code easier to understand and maintain, you also get the benefit of making documentation available within the python console, once functions are defined within the python instance. For example, once I've executed the above function definition within the python console, I can execute `help(scale01)` as follows:

```
>>> help(scale01)
Help on function scale01:

scale01(arr)
```

```
    Linearly scale the values of an array in the range [0,1]
    :param arr: input ndarray
    :return: scaled ndarray
```

Document your code (with inline comments) and provide function docstrings for each function you write in this homework ($).

Functions documented above

9. [2 points] Random Numbers:

Create a new function in hw1.py called `exercise9` (no arguments), and add the following functionality.

Set the random seed to 8 using the `seed()` function: `numpy.random.seed(seed=8)`. Use random `numpy.random.randint()` to produce 1000 throws of two (6-sided) die. (*Note:* `randint() is zero-based!`) Use the result to estimate the probability of double sixes. Report what you did and the result ($).

Now run your estimation procedure 9 more times, making sure that the random number generator is not reset. Report the results, and comment on how many times you got the same estimate as the first time ($).

Finally, set the seed to 8 a second time, rerun your estimation procedure 10 times again and report whether you get the same result as the first 10 times ($).

Explain why it is often important to have random number sequences that are not really random, and can be controlled ($).

Solution.

I wrote the following function:

```
def exercise9():
    """
    Estimate the randomness of throwing double-sixes
    :param: none
    :return: void
    """

    np.random.seed(seed=8)

    throws = 1000
    dbl6 = 0
    for _ in range(0, throws):
        (n1, n2) = np.random.randint(low=1, high=7, size=2, dtype=int)
        #print('{} {}'.format(n1, n2))
        if n1 == 6 and n2 == 6:
            dbl6 += 1

    print('Threw double-six {:.2}%'.format((dbl6 / throws) * 100))
```

Everytime I run it, I get the same result, "2.6%".

```
$ ./hw1.py
Threw double-six 2.6%
$ ./hw1.py
Threw double-six 2.6%
```

If I comment out `seed=8`, I will get different results:

```
$ for i in $(seq 1 9); do echo -n "$i: " && ./hw1.py; done
1: Threw double-six 2.7%
2: Threw double-six 2.4%
3: Threw double-six 3.4%
4: Threw double-six 2.1%
5: Threw double-six 2.6%
6: Threw double-six 2.6%
7: Threw double-six 2.3%
8: Threw double-six 3.2%
9: Threw double-six 3.2%
```

Putting back in `seed=8`, I go back to the same result:

```
$ for i in $(seq 1 10); do echo -n "$i: " && ./hw1.py; done
1: Threw double-six 2.6%
2: Threw double-six 2.6%
3: Threw double-six 2.6%
4: Threw double-six 2.6%
5: Threw double-six 2.6%
6: Threw double-six 2.6%
7: Threw double-six 2.6%
8: Threw double-six 2.6%
9: Threw double-six 2.6%
10: Threw double-six 2.6%
```

Being able to count on a "random" number allows one to write tests for such functions.

10. [5 points] Random Numbers, Vectors, Matrices, and Operations

Part 10a: Create a new function in hw1.py called `exercise10` (no arguments), and add the following functionality.

Write a short script that initializes the random number generator

python: `numpy.random.seed(seed=5)`

Followed by creating two three-dimensional column vectors using

python: `numpy.random.rand` (used in the context of code to generate the vectors)

Here we're going to be explicit about the numpy arrays having 3 rows and 1 column dimension! Represent the random variables as **a** and **b** (be sure you issue the call to set the random seed immediately before creating these variables). Print them at the terminal and copy-and-paste the result here. (If using LaTeX, use the `verbatim` environment to display). ($)

Solution 10a.

```
def exercise10a():
    """
    Print two three-dimensional column vectors
    :param: none
    :return: void
    """

    np.random.seed(seed=5)
    a = np.random.rand(3, 1)
    b = np.random.rand(3, 1)
```

```
    print(a)
    print(b)

$ ./hw1.py
[[0.22199317]
 [0.87073231]
 [0.20671916]]
[[0.91861091]
 [0.48841119]
 [0.61174386]]
```

Part 10b: Using the values of $\mathbf{a}$ and $\mathbf{b}$, compute the following and display the result two ways: (1) copy-and-paste the output (from the python interpreter/terminal; again, in LaTeX use the `verbatim` environment), (2) typeset the output (e.g., using the LaTeX math environment).

1. $\mathbf{a} + \mathbf{b} =$? ($)
2. $\mathbf{a} \circ \mathbf{b} =$? (element-wise multiply; Note: the notation $\mathbf{a} \circ \mathbf{b}$ is also known as the Hadamard product, the entrywise product, or the Schur product.) ($)
3. $\mathbf{a}^\top \mathbf{b} =$? (also called the dot-product) ($)

Solution 10b.

```
def exercise10b():
    """
    Print two three-dimensional column vectors
    :param: none
    :return: void
    """

    np.random.seed(seed=5)
    a = np.random.rand(3, 1)
    b = np.random.rand(3, 1)
    print("a\n {}".format(a))
    print("b\n{}".format(b))
    print("a + b\n{}".format(a + b))
    print("a * b\n{}".format(a * b))
    print("a . b\n{}".format(a.transpose().dot(b)))

$ ./hw1.py
a
 [[0.22199317]
 [0.87073231]
 [0.20671916]]
b
[[0.91861091]
 [0.48841119]
 [0.61174386]]
a + b
[[1.14060408]
 [1.35914349]
 [0.81846302]]
a * b
```

```
[[0.20392535]
 [0.4252754 ]
 [0.12645917]]
a . b
[[0.75565992]]
```

Part 10c: Now, set the random seed to 2 and immediately generate a random 3×3 matrix $\mathbf{X}$. In your solution, display the value of $\mathbf{X}$. Using $\mathbf{X}$ and the earlier values of a and b, compute the following in python and typeset the results in two ways, as before.

4. $\mathbf{a}^\top \mathbf{X} = \ ?$ ($)
5. $\mathbf{a}^\top \mathbf{X}\mathbf{b} = \ ?$ ($)
6. $\mathbf{X}^{-1} = \ ?$ ($)

Solution 10c.

```
def exercise10c():
    """
    Vector/matrix manipulation
    :param: none
    :return: void
    """

    np.random.seed(seed=2)
    X = np.asmatrix(np.random.rand(3, 3))
    a = np.random.rand(3, 1)
    b = np.random.rand(3, 1)
    print("a\n {}".format(a))
    print("b\n{}".format(b))
    print("X\n{}".format(X))
    print("aTX\n{}".format(a.transpose() * X))
    print("aTXb\n{}".format(a.transpose() * X * b))
    print("X-1\n{}".format(X.getI()))
```

```
$ ./hw1.py
a
 [[0.26682728]
 [0.62113383]
 [0.52914209]]
b
[[0.13457995]
 [0.51357812]
 [0.18443987]]
X
[[0.4359949  0.02592623 0.54966248]
 [0.43532239 0.4203678  0.33033482]
 [0.20464863 0.61927097 0.29965467]]
aTX
[[0.495017   0.59570483 0.51040698]]
aTXb
[[0.46669972]]
X-1
```

```
[[-1.20936675  5.11771977 -3.42333228]
 [-0.96691719  0.279414    1.46561347]
 [ 2.82418088 -4.07257903  2.64627411]]
```

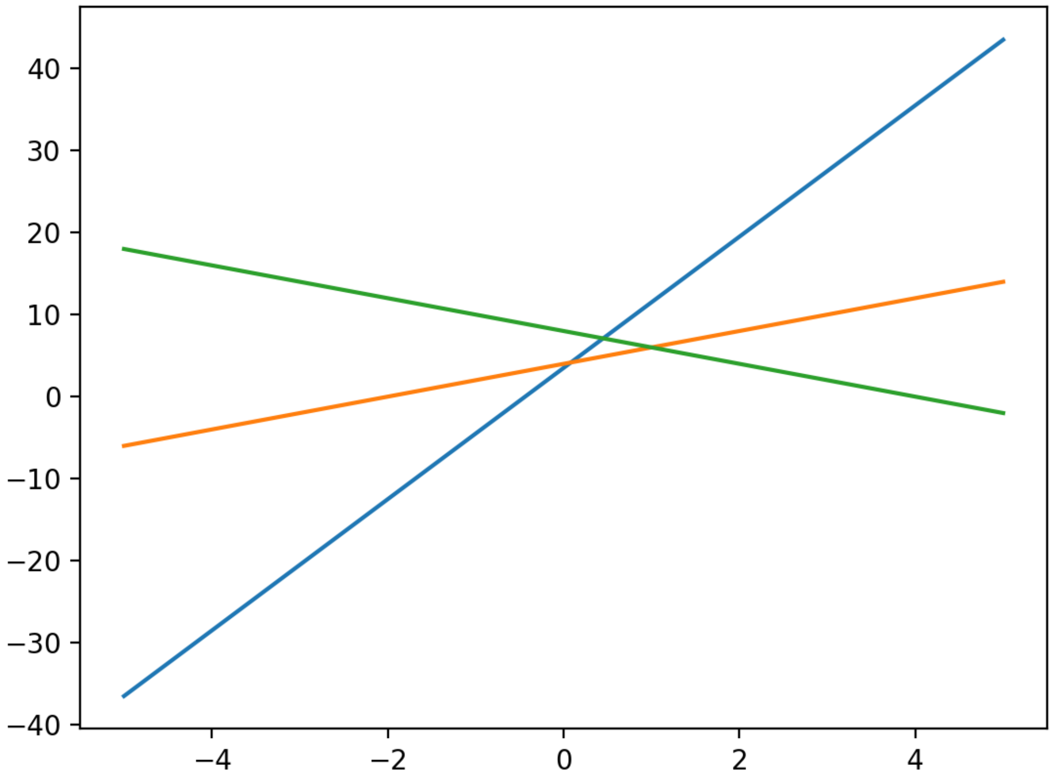

Figure 4: y = 3.5 + 8.0 x; y = 4.0 + 2.0 x; y = 8.0 + -2.0 x

11. [3 points] Simple Plotting

Use the provided python script `plotlinear.py` and plot three lines (parameters of your choosing). Place the output graphic in your pdf submission and provide a descriptive caption that indicates the intercept and slope values you used to generate the lines. ($)

Solution 11a.

I can't seem to get the figure to show here.

```
y = 3.5 + 8.0 x
y = 4.0 + 2.0 x
y = 8.0 + -2.0 x
```

Now, for another plot: create a function called exercise11 in hw1.py. Add the following functionality:

Generate a vector whose entries are the values of $\sin(\mathbf{x})$ for $\mathbf{x}$ in the range $[0, 10]$ in steps of 0.01, and plot it. Make it so that the horizontal and vertical axes of the plot are labeled and include a plot title: label the y-axis 'sin(x)', the x-axis 'x values' and provide a title for the plot, 'Sine Function for x from 0.0 to 10.0' (look at pyplot xlabel, ylabel and title). (In this course, you must **always** label you plot axes!) Include your plot in the pdf submission. (2$)

Solution 11b. <Solution goes here>

Appendix: Numpy Arrays and Linear Algebra

Here are some notes on the relation of numpy arrays to the linear algebra math of "vectors" and "matrices" we have been using. The bottom line is that vectors are, in an important sense, just limit cases of matrices that are two-dimensional, but one of the dimensions is size 1. In all code examples I will use numpy arrays as our base data structure to represent vectors and matrices.

Numpy does provide matrix objects; these build on top of the numpy.ndarray data structure. While they are nice for reducing some of the verbosity of code, we can do *everything* with plain old ndarrays. (numpy.matrix objects introduce (at least when you are first getting used to them) potential confusion about what operators are being used, due to operator overloading.) In this course, for pedagogical reasons I'll stick with numpy arrays for all representation of vectors and matrices.

There are a variety of ways to create arrays (here is a helpful overview: `http://docs.scipy.org/doc/numpy/user/basics.creation.html`). Numpy arrays can be n-dimensional. The dimensionality of an array can be accessed by the attribute `shape`, which is represented as a tuple where each position in the tuple represents the number of indices in the corresponding dimension. Here are three arrays and their shape:

```
>>> a = numpy.array([1, 2, 3])  # This creates a 1-dimensional array
>>> a
array([1, 2, 3])
>>> a.shape
(3,)  # this shows the array is 1-d with three elements in the first dimension
>>> b = numpy.array([[1, 2, 3], [4, 5, 6]])  # This creates a 2-dimensional array
>>> b.shape                   # We see this array is two-dimensional with 2 indices
(2, 3)                        # in the first dimension and 3 in the second.
>>> b                         # We generally interpret the first dimension as row
array([[1, 2, 3],             # indices, the second as column indices.
       [4, 5, 6]])
>>> b[1, 2]       # We can use the bracket notation to index into the array;
6                 # keep in mind that python indices are 0-based, so b[1, 2] is
                  # picking out the second row, third column
>>> c = numpy.array([[1], [2], [3]])  # this defines a 2-d array
>>> c.shape
(3, 1)
>>> c
array([[1],
       [2],
       [3]])
```

We can create higher dimensional arrays by nesting more lists specifying elements, but for representing vectors and matrices, we'll stick to 1 and 2-dimensional arrays. Now for the connection with the linear algebra we have used. It turns out that 1-d arrays are equivalent to column vectors: as noted in the comments, the convention is that the first dimension (even for 1-d arrays) is interpreted as indexing the rows of an array – so all 1-d array indices can be interpreted as spread across rows – i.e., their natural interpretation is as a column vector. One thing that is a little odd is that taking the transpose of a 1-d numpy array has no effect: there is only one dimension so the transpose cannot swap dimensions. For 2-d arrays, however, we can explicitly transpose, and therefore we can make an explicit distinction between column and row vectors (as in c vs. c1, below):

```
>>> a1 = a.T  # This "transposes" the 1-d array, which does nothing
>>> a1
array([1, 2, 3])
```

```
>>> a1.shape
(3,)  # this shows the transpose of a is still a 1-d array,
      # safely interpreted as a vector with three rows: a column vector
>>> b1 = b.T  # This transposes the b 2-d array
>>> b1.shape                # We now see this array is two-dimensional with 3 rows
(3, 2)                      # and 2 columns (i.e., 2 elements per row)
>>> b1
array([[1, 4],
       [2, 5],
       [3, 6]])
>>> c1 = c.T  # now we transpose c, which was a 2-d array
>>> c1.shape
(3, 1)   # c1 now has 1 row with 3 columns.
>>> c1   # This is most naturally interpreted as an explicit row vector
array([[1, 2, 3]])
```

You will see in the provided code that I generally follow the (popular) convention that as long as I am only working with a column vector that does not need to be transposed into a row vector, I will use a 1-d array. IF, however, I know that I will need to switch between column and row vector forms, then I will use the explicit 2-d array as in c and c1 (where one of the dimensions has only 1 index).

Here is a handy trick for converting a 1-d array into an explicit 2-d array as a column vector:

```
>>> a.shape    # recall that a is a 1-d array (with 3 row indices)
(3, )
>>> d = numpy.array(a[:, numpy.newaxis])  # The ‘:’ is the slice operator
>>> d.shape     # The numpy.newaxis allows us to add a new axis
(3, 1)          # (1 more dimension) to our array object ... which just
>>> d           # makes explicit that our previously "implicit" column
array([[1],     # vector is now an explicit column vector, which can then
       [2],     # be transposed...
       [3]])
```

The creation of the array d is built from indexing into the vector a, using a slice operator to refer to all elements along the first dimension, and then effectively adding a second dimension (of size 1) using the numpy.newaxis as the specifier to the second dimension index.

See here for more information about indexing:
`http://docs.scipy.org/doc/numpy/user/basics.indexing.html`
and
`http://docs.scipy.org/doc/numpy/reference/arrays.indexing.html`

You will also want to look at the numpy linalg package for linear algebra operators (esp. for the `dot` and `inv` operators):
`http://docs.scipy.org/doc/numpy/reference/routines.linalg.html`