CS 132 – Geometric Algorithms
Notes on Linear Algebra in Python

Mark Crovella

Floating Point. Python has both integer and floating point types. In python 3, dividing two integers will yield a floating point value (which is what we will always want).

You will need to understand that floating-point calculations are not exact, because real numbers are not stored exactly when converted to floating-point representation. It's not a bad idea to read `http://en.wikipedia.org/wiki/IEEE_floating_point` if you haven't had CS210 yet.

To give an illustration of how this can affect your computations, consider this example. For two floating point numbers `a` and `b`, performing the computation `a/b` may not yield exactly the true value, but rather a value that is only *very close* to the true value. So, mathematically we would expect that `b * (a/b)` should yield exactly `a`, but computationally, `a/b` is not stored exactly, so this may not happen. So in particular, `a - (b * (a/b)))` may not be exactly zero (though it will be a very small number). This will be discussed in more detail in lecture; you will need to keep this in mind throughout the course.

In general, you will often want to **treat very small numbers, eg 10^{-8} or smaller, as being like zero.** You will need to think about whether your calculation can create such small numbers in situations where zero is expected. You will need to test for and correct those situations in your code. Read the documentation for `numpy.isclose()` for ideas on how to do that.

Linear Algebra in Python. The linear algebra package we will use is `numpy`. Always import it as

```
import numpy as np
```

Matrices will always be represented as *numpy* arrays. Some important functions:

```
np.array
np.shape
np.readtxt
np.savetxt
```

Read the documentation on these before starting. Later there will be more functions from numpy that we will use. Get used to consulting the documentation!

A vector in `python` is a 1-D `numpy` array. For example, here the vector `x` has been constructed from a list of floats:

```
x = np.array([1.0, 2.0, 3.0])
```

A matrix in `python` is a 2-D `numpy` array. It behaves like a list of lists that correspond to rows. So you can construct a matrix like this:

```
A = np.array([[1.0, 2.0],[3.0, 4.0]])
```

Thus `A[0][0]` (or `A[0,0]`) is a single element of the array `A` (the element in the upper left corner), while `A[0]` is the first row of `A`. You can also use a colon (`:`) to mean "all indices." So another way to get the first row of `A` is `A[0,:]`, and to get the first column of `A`, you would use `A[:,0]`.

Numpy array types. Numpy assigns a type to each matrix – for example, `float` or `int`. If you enter a matrix by hand into the python interpreter, and all of the entries are integers, then numpy will auto-detect this as an integer matrix. When you assign values to an integer matrix they will be rounded to the nearest integer. This is **not** what you want. So you do **not** want to work with integer matrices in general.

So it is a good idea to make sure that the inputs your your functions are floating point matrices. To convert an integer matrix to a floating point matrix you can simply use:

```
A = A.astype(float)
```

Parameter Passing. Remember that parameter passing in `python` is by reference. Thus if you pass an array into a subroutine, and modify the array in the subroutine, the modifications will still be in effect after the subroutine exits (i.e., modifying an array creates side-effects).

For similar reasons, if `A` is an array, the code `B = A` does not create a new array `B`, but rather results in both `A` and `B` pointing to the same array. If you want to make an actual copy of an array or any object, use its `.copy()` method; i.e., write `B = A.copy()`.