1 Putting the process into perspective

Here is an overview of the steps we must take to learn something from data:

- **Data Collection**
  - Task: We collect/generate data.
  - Project Code: It depends upon how you found the data. Maybe you needed to write scripts to do things with it, or maybe you were able to compile it in a spreadsheet.

- **Pre-processing**
  - Task: The pre-processing is done so that visualization and analysis make sense. I.e. we may pre-scale to get the units to be comparable.
  - Project Code: Any data-specific scripts you may have written. In the future, when you have an all-encompassing data visualization/analysis program, it will need to read in the pre-processed data. You will do that with Data class in data.py. It is a design decision how you handle normalization - you may want to normalize it as part of the scaling step (i.e. before it gets loaded into Data), or you may want the Data object to handle it for you, or maybe you will want the analysis and visualization to take on the responsibility of normalization.

- **Analysis**
  - Task: To run some formal analysis such as PCA or clustering (on either the data or the results of another analysis)
We have been talking about range selection. It fits into the pipeline as part of the interactive visualization. For the most part, we will assume we are looking at all the data, and then allow the user to zoom in if they want to. You can add support for more sophisticated range selection either through pre-processing or by adding more control to your display GUI.

For the next few lectures, we will talk about pre-processing, including some visualization that will help us to improve our understanding of the data, so that we can pre-process it better. In other words, there is an arrow missing from the above diagram, from visualization back to pre-processing. (Stephanie left that arrow out of the diagram because it won’t be explicitly built in to the course project).

1.1 Pre-scaling

Interactive viewing, particularly when it involves rotation, requires that the data space axes all have the same scale. Unit distance along one axis needs to be the same as unit distance along any other axis. The mathematical difficulty of using different scales arises when we try to define the extent of an oriented bounding box as it rotates in the data space. A bounding box of unit length oriented parallel to one axis will shrink or expand when it rotates to orient with a different axis with a different scale.

Many data spaces, however, have different natural scales that are not necessarily related. A data set of height and weight contains two different natural scales (e.g. cm and kg, or inches and pounds). If we have a method of selecting an appropriate range for each axis, then we can pre-scale each axis to the range \([0, 1]\) prior to implementing the view volume.

Note that natural scale is different than range. A data set that contains the height of children might have a natural scale in inches or centimeters. A different data set that contains the height of adults will have the same natural scale, but a different range. We don’t want to use different scales when plotting these two data sets together. If we want to see their comparative values, we would also want to use the same range along each axis. Decisions about the natural scale and range are complex and data specific.

We will talk about normalization later. Normalization is putting all the data in the range 0 to 1, and may include something more sophisticated than scaling and translating. That said, for the purposes of the projects in this course, you may find that normalization and scaling are essentially the same thing.
2 Numpy

Numpy is the library that supports linear algebra operations. The two most important data structures are the n-dimensional array and the matrix. The matrix always has 2 dimensions. It is the matrix you should be using for most of your code in this course, so let’s look at some matrix operations in the Python interpreter.

First, let’s make a row vector, storing it as a matrix:

```python
>>> import numpy as np
>>> a = np.matrix([[1, 3]])
```

```latex
matrix([\begin{bmatrix} 1 \\ 3 \end{bmatrix}])
```

To make a column vector, we take the transpose:

```python
>>> b = a.T
```

```latex
matrix([\begin{bmatrix} 1 \\ 3 \end{bmatrix}])
```

But b is not a copy of a, merely another way to view the same data. We see this by changing a value of a and then looking the the contents of b (they have been affected, too). To access an element we use two indices (always use 2 indices for a matrix, even if it has only one row or only one column!). The row index is first and the column index is second.

```python
>>> a[0,0] = 4
>>> a
matrix([\begin{bmatrix} 4 \\ 3 \end{bmatrix}])
```

```latex
matrix([\begin{bmatrix} 4 \\ 3 \end{bmatrix}])
```

To decouple b from a, we make a copy of a:

```python
>>> b = a.copy().T
```

```latex
matrix([\begin{bmatrix} 4 \\ 3 \end{bmatrix}])
```

To see the shape of a, we use the shape field. This is very helpful for debugging. Instead of printing the contents of a, you can print the shape. It is a tuple: the first element is the number of rows, the second is the number of columns.

```python
>>> print a.shape
(1, 2)
```

Now lets look at slicing. First make a $3 \times 3$ matrix.

```python
>>> A = np.matrix([ [1, 2, 3], [4, 5, 6], [7, 8, 9] ])
```

```latex
\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}
```

```python
>>> A
```

```latex
\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}
```
matrix([[1, 2, 3],
        [4, 5, 6],
        [7, 8, 9]])

Now we take the third column and then the second row.

```python
>>> a2 = A[:, 2]
>>> print a2
[[3]
 [6]
 [9]]
```

```python
>>> ab = A[1, :]
...]
>>> ab
matrix([[4, 5, 6]])
```

And we try to change an entry in the row vector ab using only one index. It fails:

```python
>>> ab[1] = 10
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: index 1 is out of bounds for axis 0 with size 1
```

So, like good little programmers, we use two indices and it works. Also we see that we are changing the data in A. So, the slices are just views on the same data.

```python
>>> ab[0, 1] = 10
>>> a2
matrix([[3],
        [6],
        [9]])
>>> a2[1, 0] = 3.0
>>> a2
matrix([[3],
        [3],
        [9]])
>>> A
matrix([[1, 2, 3],
        [4, 10, 3],
        [7, 8, 9]])
```

The take-home message:

- Matrices are always 2D and you should use two indices to slice them or index into them
- Slicing and transposing a matrix does not also copy the matrix.

Acknowledgement: Most of the material in this lecture is taken from Bruce Maxwell’s spring 2012 CS251 notes.