APPM 2460
VECTORS & MATRICES I

1. Introduction

We're beginning to get in to the juice of Matlab. We've seen numerical solution to ODEs, and now we'll look at matrices, which are the dominant data structure when writing in Matlab. We'll be focusing today on how to index matrices; that is, how to obtain certain parts of matrices.

2. Review of Indexing for Vectors

Let's first recall how we index vectors. When I say "how we index" I mean how we find certain elements of a list, how we assign new values to them, and so on. Let's make a list of squares. We could type in

```
>> v = (1:10).^2

ans =

    1 4 9 16 25 36 49 64 81 100
```

So our vector `v` is just the squares of the numbers 1 through 10. If we want to look at the 4th through 6th element of this vector, we type `v(4:6)`. If we want to look at elements 2 through the end, we can type `v(2:end)`. If we want elements 1, 5, 2, and 6, in that order, we can type `v([1 5 2 6])`.

```
>> v([1 5 2 6])

ans =

    1 25 4 36
```

Note that if we type `v(:)`, Matlab returns **all** of the vector `v`.

```
>> v(:)

ans =

    1 4 9 16 25 36 49 64 81 100
```

This will become important when we begin to work with matrices.

3. Matrix basics

Now that we've reminded ourselves how things go with vector indexing, let's step up a level to matrix indexing.

3.1. **Defining matrices.** Matrix columns are separated by commas or spaces, just like vector elements, while row are separated by semicolons. here's an example: If I type in `A = [1 2 3;4 5 6;7 8 9]`, Matlab spits out:

```
A =
     1     2     3
     4     5     6
     7     8     9
```

You can see how the semicolons separate the rows of the matrix.

3.2. **Matrix indexing.** Accessing elements of a matrix is similar to a vector, except you have 2 indices. So if you typed in `A(i,j)` you would be asking Matlab for the i^{th} row and the j^{th} column. The convention is $(row, column)$, which is backwards from the (x, y) convention used for Cartesian coordinates. It's a bit confusing so just say "row column. row column. row column" over and over until you remember! Using my example matrix A, if I typed $A(2, 3)$ into the command line, Matlab would return 6, since it's in the 2^{nd} row and 3^{rd} column.

If you wanted to access the first column of A, you would do `A(:,1)` (read: all rows, column 1).

```
>> A(:,1)

ans =

    1
    4
    7
```

To access the first row, we do `A(1,:)` (read: row 1, all columns.).

```
>> A(1,:)

ans =

    1     2     3
```

We could look at multiple rows, similarly to how we look at multiple **elements** of a vector:

```
>> A(:,1:2)

ans =

    1     2
    4     5
    7     8
```

and we can mess around with ordering, like we did when we entered `v([1 5 2 6])` above

```
>> A([3 1 2], :)

ans =

    7     8     9
    1     2     3
    4     5     6
```

This is the matrix `A` with the rows reordered. We can even take slices in both directions at once:

```
>> A(2:3,1:2)

ans =

     4     5
     7     8
```

Play around with this, until you get the hang of it.

3.3. **Matrix multiplication.** Matrix multiplication is a little harder than normal multiplication, and hopefully you learned about it in class! In Matlab, matrix multiplication is performed by using the * (shift 8) symbol. As an example, try defining a matrix of all 1s, via

```
>> B = ones(3,3)
```

```
ans =

    1    1    1
    1    1    1
    1    1    1
```

Now if I multiply the matrix A that was defined above by the new matrix B, I get a strange result:

```
>> A*B
ans =

     6     6     6
    15    15    15
    24    24    24
```

Basically, the ij^{th} entry in the answer is the dot product of the i^{th} row of A and the j^{th} column of B. **TL;DR: Writing `A*B` does NOT give you the first element of `A` times the first element of `B`, and so on. Instead, `A*B` performs matrix multiplication.**

This is why we've been using `.*` or `.^2` to evaluate functions. We didn't want matrix multiplication, we wanted **element wise multiplication**. If you do A "dot times" B, the answer is quite different:

```
>> A.*B
ans =

     1     2     3
     4     5     6
     7     8     9
```

Again, element `(3,2)` of the matrix `A.*B` is `A(3,2)*B(3,2)`, and so on, unlike for `A*B`.

3.4. **Matrix indexing using for loops.** Sometimes it's useful to populate a matrix with values using for loops. Let's say you want to build a matrix whose elements were the sum of the row and column index. That means that, for example, $A(1,4) = 5$, $A(5,2) = 7$, and so on. To build such a matrix, I could write the following function:

```
function A = buildMatrix(n)

A = zeros(n,n);

for i = 1:n
    for j = 1:n
        A(i,j) = i+j;
    end
end
```

Note how I initialized A by doing `zeros(n,n)`. This is important for speed. You'll do something similar to this

4. Homework

This is the 5×5 Hilbert matrix.

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

What is the pattern for entries of this matrix? (Hint: The entries only depends on the row and column index). Using this pattern, write a function, `myHilb.m`, to construct an $n \times n$ Hilbert matrix. Write a script that shows the function in action, for a few different values of n.