14:440:127– Introduction to Computers for Engineers

Notes for Lecture 03

Rutgers University, Spring 2010 Instructor- Blase E. Ur

1 Strings

Character strings do something interesting when you try to put them into a vector. The following example motivates our exploration of the subject:

```
a = 'hello';
b = 'goodbye';
c = [a b]
     c = 'hellogoodbye'
c(2) = 'e'
```

As you see, placing two strings in a vector effectively combines them into one long string. This is because strings themselves are actually vectors of single characters, and thus we are combining two shorter vectors into one long vector. Since character strings in Matlab are simply vectors of single characters, any operation we perform on vectors can also be used on a string:

```
a = 'I love Matlab';
a(3:6) = 'hate'
     ans = 'I hate Matlab'
a((end-2):end) = []   % deletes those values
     % by setting them to the 'empty vector'
     ans = 'I hate Mat'
fliplr(a)
     ans = 'taM etah I'
```

Also note that to include an apostrophe in a string, use two single quotes next to each other i.e. `'matlab''s stupid'`

Now what if we wanted to create a data type analogous to a vector, but for strings?

1.1 Cell Arrays

A *cell array* is a type of array (matrix) in which each element can be a vector or matrix itself. Since character strings are vectors, a cell array essentially allows us to make a 'pseudo-vector' of strings.

However, the syntax for a cell array differs slightly from a vector. Rather than using square brackets to define a cell array, we instead use squigly braces– `{ and }`. To access individual elements, we again use squigly braces rather than parentheses. However, to copy/change/move parts of the cell array, we use parentheses as normal. Here's an example:

```
a = { 'one' 'two' 'three' }; % squigly braces
b = a(2:3);  % note that parentheses are used
             % this is because we are copying parts of the array
fprintf('The number is %s \n', b{2} )  % note squigly braces
       The number is three
```

1.2 String Comparison- strcmp

Because strings are actually vectors of characters, funny things happen when you try to compare two strings using the == double equals signs. If the strings are the same length, this works (giving you a vector of 1's and 0's as the

result). However, if the strings aren't the same length, you get an error.

To compare strings more easily, there's a function called *strcmp*– string comparison. If returns either 1 or 0 (true or false) whether the strings are the same, case sensitive.

```
thislecture = 'blah blah blah';
if(strcmp(thislecture,'brilliant insights'))
    disp('job well done')
else
    disp('waste of tuition money')
end
```

2 Element by Element Math Operations on Matrices

Mathematical operations on Matrices can be a bit tricky since there are multiple ways in which operations such as multiplication can be defined mathematically on matrices. However, other operations such as addition and subtraction are much simpler since they operate 'element by element.'

2.1 Add, Subtract

In order to add or subtract two vectors or matrices, they must have the same dimensions (number of rows, number of columns). The element in the first row and first column of the first matrix is added to the corresponding element in the second matrix, and that becomes the element in the first row and first column of the result:

```
a = [ 7 2 ; 6 3 ];
b = [ 1 3 ; 1 5 ];
a + b
    ans = [ 8 5 ]
          [ 7 8 ]
```

2.2 Element by Element (Dot) Multiply, Exponentiate

The multiplicative analogue of addition and subtraction, which work 'element by element,' is called 'dot multiplication' and is defined using a period followed by a multiplication sign– `.*`

```
a = [ 7 2 ; 6 3 ];
b = [ 1 3 ; 1 5 ];
a .* b
    ans = [ 7  6 ]
          [ 6 15 ]
```

Similarly, to raise each individual element in a matrix to a particular power, which indeed is just repeated multiplication, also uses a dot operator– `.^`

```
b = [ 1 3 ; 1 5 ];
b.^2
    ans = [ 1  9 ]
          [ 1 25 ]
```

3 Functions for Matrices and Vectors

3.1 Sum, Prod

Matlab includes functions that add or multiply all of the numbers in a vector. As you might guess, inputting a vector to the *sum* function adds all of the elements of the vector, whereas the *prod* function multiplies all of the elements.

```
sum( [ 1 5 7 ] )
                ans = 13
prod( [ 2 5 7 ] )
                ans = 70
```

Note that applying these functions to a matrix results in summing or multiplying *EACH COLUMN* of the matrix and outputs a vector containing the sum or product of each column. Thus, to sum all of the elements of a matrix *M*, you first run *sum(M)* to create a vector of the sums of each column, and then sum that result: *sum(sum(M))*.

```
M = [ 1 5 7 ; 6 1 2 ]
   ans = [ 1 5 7 ]
         [ 6 1 2 ]
sum(M)
   ans = [ 7 6 9 ]
sum(sum(M))
    ans = 22
```

3.2 Max, Min

Matlab also provides functions to find the maximum and minimum values of a vector or matrix. These are the creatively named *max* and *min* functions. For vectors, they return a single scalar with the absolute maximum number. For matrices, they return the maximum or minimum number *in each column*. Thus, as with the *sum* and *prod* functions, to find the absolute largest number in a matrix *M*, you would use *max(max(M))*.

The *max* and *min* functions can also be used to identify the location of the maximum or minimum number inside a vector (or, with a little bit more trouble, a matrix). If you write a statement like *[a,b] = max(V)*, where *a* and *b* are names of variables and *V* is a vector, then *a* will contain the largest value and *b* will contain the location of that largest value inside the vector V. Note that [a,b] is essentially just a vector of two variables. Also note that if the largest value is found more than one time inside the vector, *b* will contain the first location where it is found.

```
V = [ 5 7 6 1 2 ];
[a,b] = min(V)
     a = 1   % The smallest element is 1
     b = 4   % 1 is the 4th element of V
```

What if you wanted to find the row and column in which the maximum or minimum is located? Here, you can be creative using a few different methods, one of which is shown:

```
V = [ 5 7; 6 1; 2 3];
[bigC,rowsC] = max(V)
     bigC = [6 7]   % The largest element IN EACH COLUMN
     rowsC = [2 1]   % The ROWS that contain the largest elements
[big,column] = min(bigC)
     big = 7   %  The largest overall element
     column = 2   % The COLUMN containing the overall largest element
row = rowsC(column)  % Column 2 contains the largest overall element.
          % Check which ROW in Column 2 contains the largest element.
     row = 1
```

3.3 Mean, Median

The *mean* and *median* functions operate on vectors just as you'd expect from statistics. The *mean* calculates the average, the sum of the elements of a vector divided by the number of elements. The *median* identifies the middle number if the vector were to be ordered from smallest to largest element.

As with the other matrix functions, calculating the mean or median of a matrix calculates the mean or median of each column. Thus, to find the mean of a matrix *M*, use *mean(mean(M))*. However, this wouldn't work for the median– think about why! Instead, use *mean(M(:))*, which first converts the matrix *M* to a vector (by making a

vector containing ALL of the elements) and then finding the median.

3.4 Sort, Sortrows

You can also sort the elements of a vector in ascending (smallest to largest) order by using the *sort* function. To instead sort a vector V from largest to smallest, instead use *flipud(sort(V))* for column (vertical) vectors and *fliplr(sort(V))* for row (horizontal) vectors. For matrices, note that the *sort* function sorts each column individually.

There is a similar function called *sortrows*, but be careful– it doesn't sort the rows individually and is thus not an analogue of the *sort* function. Instead, it merely performs a sort in which the rows are kept together and then ordered beginning with the elements in the first column, and then the second, and so on. If the rows actually just contain words (i.e. you have characters as each element), *sortrows* would *alphabetize* the words.

3.5 Transpose

The transpose operation switches the rows and the columns of a matrix. What was the first row now becomes the first column. What was the second row now becomes the second column. The transpose operation can actually be performed in two ways in Matlab: using an apostrophe (single quote) following the name of a variable or a matrix, or using the *transpose* function.

```
A = [ 1 5 ; 7 9 ]
    A =   [ 1 5 ]
          [ 7 9 ]
A'
    ans = [ 1 7 ]
          [ 5 9 ]
transpose(A)
    ans = [ 1 7 ]
          [ 5 9 ]
```

Transpose is actually a very useful operation mathematically, as well as in signal processing (consider flipping a photograph that's represented as a matrix of pixels.)

3.5.1 Actually Sorting Rows

We know that the *sort* function sorts each column of a matrix individually. What if we wanted to sort each row individually instead? We could follow this algorithm:
1. Transpose the matrix (i.e. switch the rows and the columns).
2. Sort this transposed matrix. The columns it is sorting are the rows of the original matrix.
3. Transpose this sorted matrix so that the rows are again the rows and the columns are again the columns.

Thus, the following equivalent lines of code sort the rows of a matrix M:

```
transpose(sort(transpose(M)))  %option 1
sort(M')'   %option 2
```

3.6 Flipping a Matrix

To flip a matrix from left to right, use the *fliplr* function. To flip the matrix from up to down (i.e. the first row becomes the last row), use *flipud*.

3.7 Calculating Determinants

To calculate the determinant of a matrix in Matlab, use the *det* function. Recall from math class that when the determinant equals 0, there is no inverse for that matrix (and it is said to be singular). On a high level, the determinant is the sum of the products of the top left to bottom right diagonal columns, minus the sum of the products of the top right to bottom left diagonal columns.

```
A = [ 1 5 ; 7 9 ]
det(A)
                ans = -26
```

3.8 Inverse

The inverse of a matrix A, notated A^{-1}, is defined such that $A * A^{-1}$ equals the identity matrix. As with transpose, there are two methods of obtaining a matrix's inverse in Matlab: the *inv* function or raising the matrix to the -1 power.

```
A = [ 1 5 ; 7 9 ]
    A =   [ 1 5 ]
          [ 7 9 ]
inv(A)
    ans = [ -0.3462    0.1923 ]
          [  0.2692   -0.0385 ]

A^(-1)
    ans = [ -0.3462    0.1923 ]
          [  0.2692   -0.0385 ]
```

3.9 Turn A Matrix into A Vector

In order to take a matrix A and turn it into a vector, simply use A(:), which means 'take all of the elements of the matrix A.'

```
A = [ 1 5 ; 7 9 ]
     A = [ 1 5 ]
         [ 7 9 ]
B = A(:)
     B = [ 1; 7; 5; 9 ]
```

4 Filtering And Choosing Certain Elements

Being able to pick particular elements out of a matrix or vector is very useful. The first key in filtering matrices and vectors is to understand how Matlab interprets logical operators involving a scalar (single value) when applied to matrices. For a matrix M, when given an operation like $M > 5$, Matlab returns a matrix the same size as M, but filled with 1's and 0's (true and false) indicating whether or not that statement was true for each element of the matrix.

```
A = [ 1 5 ; 7 9 ]
     A = [ 1 5 ]
         [ 7 9 ]
A>=5
     ans = [ 0 1 ]
           [ 1 1 ]
```

4.1 Logical Indexing

Interestingly (and rather uniquely among program languages), Matlab allows you to use these logical (true and false) matrices to index (or choose) elements from a matrix or vector. Given a logical matrix as the index, Matlab returns a vector of only those elements for which the corresponding element in the logical matrix was true:

```
A = [ 1 5 ; 7 9 ]
     A = [ 1 5 ]
         [ 7 9 ]
A(A>=5)
     ans = [ 7; 5; 9 ]

A(A>=5) = 99
     A = [  1 99 ]
         [ 99 99 ]
```

4.2 Find

Whereas logical operators by themselves return a 1's/0's matrix (true and false) and whereas using logical indexing returns only the elements of a matrix for which the condition is true, the *find* function returns a **vector of the LOCATIONS**, indexed with a single number, of the elements in a matrix that meet a certain condition. In other words, find will return a vector containing the indices of all matrix elements for which that conditional is true.

```
A = [ 6 3 8 2 1 ];
locations = find(A>5)
                locations = [ 1 3]
```

This means that the 1st and 3rd elements of A are greater than 5. To return vectors of the rows and columns, respectively, of the elements meeting a certain condition, simply say something like `[rows cols] = find(V==5)`

4.3 More Examples

Let's say we had a matrix M and wanted to sum every number over 100. The following code would do that:

```
sum(M(M>100))
```

Note that although the 'M(M...' part of this example seems redundant, it certainly is not. Moving from the innermost parentheses outwards, 'M¿100' returns a logical (true and false / 1 and 0) matrix. Thus, '$M(M > 100)$' returns all elements of M that are greater than 100. This is a vector, so we just sum that vector to get the answer!

Here's one more example of how you can use the outputs of different functions to feed into each other. Let's say we had a matrix *M = [285 146 137 ; 69 267 6 ; 182 229 246]* and we wanted to find the largest number between 130 and 150. We could use the following code:

```
M = [ 285 146 137 ; 69 267 6 ; 182 229 246 ];
max(M(M>=130 & M<=150))
```

Note that `M(M>=130 & M<=150)` will be a vector containing [146 137], the two numbers fitting the criteria of being between 130 and 150, inclusive. Thus, we're just finding the max of the vector [146 137].

You've probably noticed by now that Matlab gives you lots of basic tools; however, you often have to combine them creatively to get the answer. Let's say I asked you for the sum of all the prime numbers from 103 to 1003, inclusive. Well, you know that the *sum(primes(x))* gives you the sum of the primes from 1 to x. Well, why not just sum all the primes from 1 to 1003, and subtract the sum of all the primes from 1 to 102... that leaves you with the sum of the primes from 103 to 1003:

```
% sum the primes from 103 to 1003
sum(primes(1003) - sum(primes(102))  % 102 is 1 less than 103
```

Now, what if I wanted you to just create a vector of the primes from 103 to 1003? Well, we can create a vector of all the primes from 1 to 1003 and use logical indexing:

```
% create a vector V of the primes from 103 to 1003
a = primes(1003);
V = a(a>=103);

% Here's an alternate, less elegant, solution
a = primes(1003);
V = a((length(primes(102))+1):end);
```

4.4 Meshgrid

The *meshgrid* function will come in handy when you least expect it. *Meshgrid* takes two vectors as arguments and creates two matrices as its output that have opposing gradients. For instance:

```
[x y] = meshgrid(1:4, 2:4)

    x = [ 1 2 3 4 ]
        [ 1 2 3 4 ]
        [ 1 2 3 4 ]

    y = [ 2 2 2 2 ]
        [ 3 3 3 3 ]
        [ 4 4 4 4 ]
```

Why is that useful? Well, if you calculate $x. * y$, you now have every possible product of an element from the first vector and the second vector using element-by-element math operations.

As an example problem, what if I asked you what values of x and y, where each is an integer from 1 to 10, maximizes the expression $z = cos(x)sin(y)$? Well, use *meshgrid*:

```
[x y] = meshgrid(1:10, 1:10);
z = cos(x).*sin(y);  % don't forget dot times to do element by element
location = find(z == max(max(z)));
fprintf('The maximum occurs when x is %.0f and y is %.0f\n',x(location),y(location))
```

4.4.1 Pythagorean Triplet Example

(This question is taken from ProjectEuler.net) A Pythagorean triplet is a set of three natural numbers, $a < b < c$, for which $a^2 + b^2 = c^2$. For example, $3^2 + 4^2 = 9 + 16 = 25 = 5^2$. There exists exactly one Pythagorean triplet for which a + b + c = 1000. Find a and b.

This is one of many examples of a problem where the *meshgrid* function is unexpectedly useful. Let's use the *meshgrid* function to create corresponding matrices of the numbers from 1 to 997 (since 1 and 997 are the smallest and largest numbers for a and c, respectively, with which a+b+c could equal 1000). We then create a matrix for the values of c, again corresponding to a and b, and find where a+b+c sums to 1000.

We don't need to worry that c contains many non-integer values since a and b are both integers. Since a and b are integers, a+b+c will only sum to 1000 if c is also an integer. Since we want to find a unique solution to this problem, we've arbitrarily decided that a will be the smaller number and thus find the location where $a < b$. As the matrices a, b, and c correspond, we look at each vector in the same location:

```
[a b] = meshgrid(1:997,1:997);
c = sqrt(a.^2 + b.^2);
location = find((a+b+c)==1000 & a<b);
a(location)
    ans =   200
b(location)
    ans =   375
```

5 Mathematical Uses of Matrices

Using vectors and matrices to represent various mathematical or real-world qualities allows us to perform useful calculations using Matlab:

5.1 Matrix Multiplication

Using the multiplication symbol * by itself on matrices performs matrix multiplication, which you'll likely remember from pre-calc or linear algebra. When you perform *a*b*, the element in the first row and second column of the result is effectively the sum of each element of the *first row* of the *a* matrix multiplied by its corresponding element in the *second column* of the *b* matrix. You probably remember how tedious it is to do matrix multiplication by hand.

```
a = [ 7 2 ; 6 3 ];
b = [ 1 3 ; 1 5 ];
a * b
   ans = [ 9 31 ]
         [ 9 33 ]
```

Remember that the 'inner dimensions' of the two matrices being multiplied must be the same (i.e. the number of columns in the first matrix must equal the number of rows in the second matrix). Multiplying a matrix that is 5x3 by a matrix that is 3x6 results in a matrix that is 5x6. Trying to multiply a matrix that is 5x3 by another matrix that is 5x3 results in an error, since the inner dimensions (3 and 5) are unequal.

5.2 Dot (Scalar) and Cross (Vector) Products

Matlab also has built-in functions to compute the dot product and cross product, two forms of vector multiplication that are extremely common in physics and engineering. Recall that given two vectors *a = [a1 a2 a3]* and *b = [b1 b2 b3]*, the dot product is defined as (a1*b1)+(a2*b2)+(a3*b3), or as the product of the magnitudes of the two vectors times the cosine of the angle between them. In physics, the dot product is used to calculate Work, among other quantities.

To calculate the dot product in Matlab for two vectors *a* and *b*, simply use the *dot* function:

```
a = [ 7 2 3 ];
b = [ 1 3 5 ];
dot(a,b)
                ans = 28
```

Note that the dot product is merely *sum(a.*b)*.

On the other hand, the result of taking the cross product of two vectors is another vector. The cross product is used in physics to find the moment about a point, for instance. The cross product, for two vectors *a = [a1 a2 a3]* and *b = [b1 b2 b3]*, equals (a2*b3-b2*a3)i + (a3*b1-a1*b3)j + (a1*b2-a2*b1)k. To calculate the cross product in Matlab, use the *cross* function:

```
a = [ 7 2 3 ];
b = [ 1 3 5 ];
cross(a,b)
                ans = [ 1 -32 19 ]
```

5.3 Roots/Zeros of a Polynomial

To find the roots/zeros of a polynomial ($a*x^n + b*x^{n-1} + ... + y*x + z$), the values of x for which the polynomiual evaluates to 0, you can use the *roots* function. The trick is that you represent the coefficients of the polynomial as a vector, from the highest power to the lowest power. Thus, to find the roots of $5x^3 - 6x + 2$, you would type:

```
roots([5 0 -6 2])
    ans = -1.2345
           0.8560
           0.3785
```

This means that x=-1.2345, x=0.8560, and x=0.3785 are the roots (or zeros) of that polynomial.

5.4 Solving a System of Equations

Finding the solution to a system of equations (i.e. 3 equations, 3 unknowns) is quite easy in Matlab.
Given Example System 1:

```
x+5y+2z = 2
2x+5y+2z = 4
-6x-2y-z = -6
```

If you create a matrix A containing the coefficients of x,y,and z, this will be a 3x3 matrix:

```
A = [ 1 5 2 ; 2 5 2 ; -6 -2 -1 ];
```

Now, create a vector of the results of each equation, and save this as vector B. Be sure this is a column vector:

```
B = [ 2 ; 4 ; -6 ];
```

Now, consider writing the system of equations based on these matrices. You will see that A*x = B, where $x = [x_1; y_1; z_1]$, the values of x, y, and z.

From linear algebra, you can solve for x by multiplying by the inverse matrix: $A^{-1} * A * x = A^{-1} * B$. The result will be a vector containing the values of x, y, and z, in that order. Unfortunately, certain types of matrices will cause excessive round-off errors when the inverse is taken. Therefore, this method is shown primarily for pedagogical purposes since this is the notation commonly used in math classes.

Using the left division operator (\ ) is the preferred method of solving systems of equations. Left division is defined for matrices in Matlab such that A\B equals $A^{-1} * B$. However, in defining the operator, Matlab avoids some rounding error pitfalls. Recall that multiplication and division with matrices are not commutative, hence the need to define left division. Thus, the following series of commands solves the example system of equations above:

```
A = [ 1 5 2 ; 2 5 2 ; -6 -2 -1 ];
B = [ 2 ; 4 ; -6 ];
A\B
                ans =
     2.0000
    12.0000
   -30.0000
```

This means that x = 2, y = 12, and z = -30.

6 Sound Processing

Representing sound waves digitally involves a trick called sampling, which is a process by which the amplitude of the sound wave is recorded (or sampled) once every specified interval. For a CD, this is 44.1khz (44,1000 times each second). If you don't have enough samples per second, you can no longer reconstruct the original signal from the samples. If you're interested in this, take Signals and Systems or other Electrical Engineering classes and you'll become best buds with Claude Shannon.

Representing sound in Matlab is quite easy; all of the samples of the amplitudes are stored in a vector. The only other necessary piece of information is the sampling rate, which is how many samples are taken each second.

Let's first just play two seconds of the note A (440 hz). Note that hertz (abbreviated hz) means cycles per second. Recall that $y = sin(2 * pi * x)$ features one full cycle of the sine wave each second. Thus, $y = sin(440 * 2 * pi * x)$ will cycle 440 times each second; in other words, it's 440 hertz. If y is the amplitude of the sound wave, we need to define x as the time so that we can calculate y. Since we want 2 seconds worth of sound, sampling 44,100 times a second, we use *linspace* to equally space out 44,100 points a second over 2 seconds:

We then give the vector of the *amplitudes of the samples* and our sampling rate to a function called *audioplayer*, which returns an 'audioplayer object' that we store in the variable *p*. Now, we can *play(p)* or *stop(p)*.

```
t = linspace(0,2,44100*2+1);
  % note:  this is 44,1000 samples per second,
y = sin(440*2*pi*x)
p=audioplayer(y,44100)
  % note that we specify 44,100 samples per second
play(p)
```

Of course, playing single notes is kind of boring. Thankfully, we can import sound into Matlab using the *wavread( )* function, passing it just the file name of a wave file: *wavread('gaga.wav')*. It reads in a vector of all of the y values, the amplitudes. (You don't need the x's since they're evenly spaced, you just need to know how many y points, or SAMPLES, per second were taken). If you ask for a second output, *wavread* also returns the sampling rate of the sound file. Note that *wavread* requires a '.wav' file. (Wav files are uncompressed sound files, whereas mp3s are compressed; you can convert between the two using free software such as 'LAME.')

```
[y f] = wavread('gaga.wav');
   % y is a vector of the amplitudes of the samples
   % f is the sampling rate (often 44100)
p=audioplayer(y,f)
play(p)
```

6.1 Tricks with Sound Processing

Using vector and matrix functions, we can do quite a bit of sound processing. First, let's think about how to play a sound file backwards. Quite simply, we just need to reverse the vector that stores the samples. Noting that *wavread* gives us a column vector, we use the *flipud* function:

```
% Play backwards
[y f] = wavread('gaga.wav');
p=audioplayer(flipud(y),f)
play(p)
```

We can also speed up a sound file. One way to play a file at double speed is to trick Matlab into using twice as many samples per second as it should, effectively crunching all of the data into half of the time (and thus double the frequency). Similarly, we could use only half of the samples, which we show in option 2 below:

```
% Double Speed, Option 1
[y f] = wavread('gaga.wav');
p=audioplayer(y,f*2)
play(p)

% Double Speed, Option 2
[y f] = wavread('gaga.wav');
p=audioplayer(y(1:2:end),f)
play(p)
```

We can also create fade-outs very easily. To create a 'linear' fade, we could multiply the first sample by 1, the second sample by a little less than one, the middle sample by 0.5, the second to last sample by just over 0, and the final sample by 0. In other words, we'll multiply all of the samples by a vector that looks something like [1 .999 .998 ... .002 .001 0]. Since we know that these numbers should be evenly spaced, begin at 1, and end at 0, we use linspace:

```
% Fade out
[y f] = wavread('single.wav');
y2 = y(1:(5*f));  % Take the first 5 seconds
p=audioplayer(y2.*linspace(1,0,length(y2))',f);
play(p)
```

6.2 Sound Filtering

Here's a good tutorial (section 10.3): http://www.aquaphoenix.com/lectures/matlab10/

Let's look at some more advanced ideas with sound processing. After reading in a file using *wavread* (here, note that we automatically find out the sampling rate and save it in the variable f), we can filter the file. Filtering allows us to change some of the frequencies that comprise the sound waves. In the example below, we read in a file called gaga.wav, filter out the low frequencies below 1000 hz using what's called a high-pass filter, which only allows the *high frequencies to pass*. The function to do filtering here is called *butter*, which is short for butterworth filter (after Stephen Butterworth, who first described it).

```
[y f] = wavread('gaga.wav');
[b a] = butter(10,1000/(f/2),'high');
y2 = filtfilt(b,a,y);
p = audioplayer(y2,f);
play(p)
```

7 Images

7.1 Grayscale Images

Matlab is very good at processing images. Why? Images are made up of pixels, tiny little squares that each have an intensity (how bright it is). In the simplest case, let's look at a grayscale image. You can think of this image as a matrix where each element is an intensity (0=black, 64=white, in between=grey). You can use the *imagesc* function, where the input is a matrix, to display an image.

```
x = [64 0 64; 64 64 64; 0 25 35 ];
imagesc(x); colormap gray
```

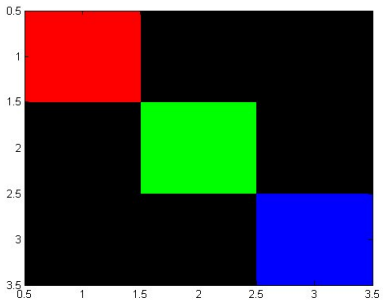
You need to specify 'colormap gray' so that it shows up in shades of gray. Otherwise, Matlab has a default colormap that uses all of the colors of the rainbow, mapping each specific number to a color. To see a number of useful colormaps, you can type 'help graph3d'.

Often, instead of *image()*, you'll want to use *imagesc()*, which scales the image so that it uses all values over the colormap.

7.2 Color Images

In contrast, color images are a 3-D matrix, split into 3 layers. Each value can range from 0 to 1, 0 to 64, or 0 to 255, depending on the type of image read in. What is a layer? Well, imagine a 3-D matrix. You have rows, columns, and then layers. Alternately, you can consider the layers to be matrices stacked on top of each other. If you store a color image in matrix *b*, b(:,:,1) is the red layer, b(:,:,2) is the green layer, and b(:,:,3) is the blue layer; like your television, Matlab works in 'RGB' (Red, Green, Blue) in that order. Having a 1.0 in only, say, the first layer means "make that pixel red."

```
a = zeros(3,3,3);
a(1,1,1)=1;
a(2,2,2)=1;
a(3,3,3)=1;
image(a)
```



Note that Matlab has built-in functions to read and write image files. You could type *a=imread('blase.jpg')* to read the file blase.jpg into the matrix a. *imwrite()* is the corresponding function to write to a file.

8 Toolboxes- The Image Processing Toolbox

Here's a good tutorial (section 10.2): `http://www.aquaphoenix.com/lectures/matlab10/`
or try `www.math.hkbu.edu.hk/~cstong/sci3710/filter_tutor.html`

In Matlab, a lot of the advanced functionality comes with what are called Toolboxes, which are basically add-ons to Matlab that cost more. Luckily, the Image Processing Toolbox is included with the student version of Matlab, allowing you to do all sorts of image enhancement and image processing. You use the *fspecial* function to create a filter, and then either *filter2* to apply the filter to a grayscale image, or *imfilter* to apply the filter to a color (RGB) image. Here's an example of code that blurs the black and white image *bw.jpg* and the color image *color.jpg*.

```
originalbw = imread('bw.jpg'); % grayscale
h1 = fspecial('motion');
newbw = filter2(originalbw,h1);
figure(1); imagesc(newbw)
figure(2); imagesc(originalbw)

originalcolor = imread('color.jpg'); % color
h2 = fspecial('motion');
newcolor = imfilter(originalcolor,h2);
figure(1); imagesc(newcolor)
figure(2); imagesc(originalcolor)
```

Try replacing *'motion'* with *'gaussian'*, or *'sobel'*, or any of a number of different filter names (see the tutorials, or Google "matlab image processing", for more ideas). Motion adds motion blur, Gaussian adds a gaussian blur, and Sobel highlights the edges (areas of the photo with rapid changes in intensity/color).

Some of the filters can take extra arguments i.e. you could call the gaussian filter with *fspecial('gaussian',[7 7],5)*, which blurs even more by specifying a 7x7 filter with a standard deviation of 5.

There are many, many toolboxes out there; try looking around in *help* to see some options.