14:440:127– Introduction to Computers for Engineers

Notes for Lecture 4- Exam 1 Review

Rutgers University, Fall 2009 — Instructor- Blase E. Ur

1 Images

- Key Idea 1: A grayscale image is a 2-D matrix where each matrix element represents the intensity (whiteness) of that pixel. Using *image*, black is 0, white is 64, and everything in between is a shade of gray. Using *imagesc*, the smallest number is black, the largest number is white, and everything in between is scaled accordingly. Using either function to display the image, make sure to say *colormap gray*;
- Key Idea 2: `x = imread('marksanchez.jpg')` reads the image marksanchez.jpg into the matrix x.
- Key Idea 3: Color images are 3 dimensional matrices, which have rows, column, and layers (1 = red, 2 = green, 3 = blue. RGB, get it?). The values for each pixel range from 0 to 1.
- Key Idea 4: For a color image, the following code applies a motion blur and displays the blurred image. If you replace 'motion' with 'gaussian', you get a gaussian blur. Replace it with 'sobel', and you find the edges of an image.

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

2 Sound System (gonna bring me back up)

- Key Idea 1: Sound is represented as a vector of sound wave amplitudes. These amplitudes are 'sampled' every specified period i.e. 44.1khz (44,100 times a second) for most audio cds.
- Key Idea 2: `[y f] = wavread('soulja.wav');` reads in a wav file. *y* is the vector of amplitudes. *f* is the sampling rate (a single number).
- Key Idea 3: To play sound, create an audioplayer object i.e. `p = audioplayer(SOMEVECTOR, THESAMPLINGRATE)` and then say *play(p)* or *stop(p)*
- Key Idea 4: You can filter i.e. high pass (only high sounds can pass through the filter) or low pass (only low, bassy sounds can pass through the filter) as follows. This is a high pass filter

with a 1000 hz cut-off frequency.:

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

3 Review- Basics of Matlab

- Key Idea 1: The names of m-files and of variables cannot contain spaces, must start with a letter, and can only contain letters, numbers, and the underscore.
- Key Idea 2: The *clear* command erases the contents of all variables that are currently set. In contrast, the *clc* command gets rid of all outputs on the screen, but doesn't change any variables.
- Key Idea 3: The first place Matlab looks for an m-file that you refer to is in your **current directory**. The next place it will look is going down the list of directories in your **path**. On the exam, you'll probably want to set the **current directory** to be the Windows Desktop. We'll show you how to do this (the exam accounts block the normal way).

4 Review- Relational and Logical Operators

- Key Idea 1: The relational operators are `>` , `>=` , `<` , `<=` , `==` , `~=`. Notice that those last two operators signify "is it equal to" and "is it UNequal to", respectively. Each of these operators returns either 0 (false) or 1 (true).
- Key Idea 2: 0 is false. 1 and *all other non-zero values* are true.
- Key Idea 3: To connect two or more statements, each of which is true or false, use the logical operators: `&` (AND) is true only if the statement before it AND the statement after it are both true. `|` (OR) is true if either the statement before it OR the statement after it is true. `~` (NOT) reverses true and false. `~1` is false, whereas `~0` is true.
- Key Idea 4: If you use relational operators with a vector/matrix and a scalar (single value), you get a logical vector/matrix (full of 1's and 0's) back. For instance, if you have a 1x15 vector V, and say $V >= 15$, you get back a 1x15 vector full of 1's and 0's, identifying whether each element meets that condition. If you use relational operators with two vectors or matrices of equal size, you again get back a vector/matrix of 1's and 0's.

5 Review- Conditional Statements

5.1 If

- Key Idea 1: Conditional statements let you choose a course of action from a bunch of possibilities by testing whether statements are true or false.

- Key Idea 2: An If Statement begins with *if(condition)*, where the condition evaluates to true or false. If the condition is true, Matlab executes the statements following the *if* and preceding *end*. If the condition is not true, Matlab skips those statements.
- Key Idea 3: If the first condition is not true, you can follow with an *else* statement. You can't write any condition after *else*. If the **original** condition is false, Matlab executes the statements following *else*.
- Key Idea 4: If you want to have a bunch of possible courses of action, you can have *elseif* statements. First, you'll have an *if* condition, then 0 or more *elseif* conditions, and then you might (or might not) have an *else*. Going down the list of conditions, Matlab executes the statements following the first condition that is true, and it **skips the rest**. There is only a single *end*, all the way at the bottom.

```
x = input('Enter an integer   ');
if(x~=fix(x))
     disp('You didn't enter an integer')
elseif(x<0)
     disp('You entered a negative integer')
elseif(x==0)
     disp('You entered zero')
elseif(rem(x,2)==0)
     disp('You entered a positive, even integer')
else
     disp('You entered a positive, odd integer');
end
```

5.2 Switch

- Key Idea 1: Instead of using *if* statements, if you have a single variable that takes on a small number of discrete values, you can use *switch case*.
- Key Idea 2: Start off by typing *switch VARIABLEname*.
- Key Idea 3: Then, for each value, type *case VALUE* i.e *case 5*. Don't put something like `case x==5`, you've already specified the variable in the *switch* statement. That would give you an error.
- Key Idea 4: An *if* statement uses *else* to capture all other possibilities. A *switch* statement uses *otherwise*.
- Key Idea 5: End a *switch case* statement with *end*.
- Key Idea 6: To test a couple of possible values in a single case, surround those values by squigly braces, i.e. `case {5,6,7}` will test if the variable is 5, 6, or 7.

```
school = input('enter your school number ');
switch school
  case 14
    disp('engineer.  the best')
  otherwise
    disp('LOSER!')
end
```

6 Review- Built-In Math Functions

Matlab includes many built-in functions for math operations. Here are a number of the most important ones:

```
sqrt(5)   nthroot(27,3)    sin(pi)   sind(75)   log(5)   log10(5)   exp(5)
rem(15,2)   factor(15)   factorial(15)    primes(100)   isprime(101)
round(5.3) % (.5 or greater rounds towards greater magnitude)
fix(5.3)  % towards 0
floor(5.3) % towards -inf       ceil(5.3) % towards +inf
```

- Key Idea 1: The trig functions assume input in radians. Use the 'd' functions (i.e. sind( )) if the input is in degrees.
- Key Idea 2: The *log* function is the natural log, base e. The logarithm base 10 is the *log10* function.
- Key Idea 3: The *factor* function returns a vector of the primes factors of the input number. The *primes* function returns a vector of all prime numbers between 1 and the input number. The *isprime* function returns, true or false, if the input number is prime.

7 Review- Inputs / Outputs

- Key Idea 1: The *input* function lets you get input from a user. A typical call would be *someVariable = input('Display this on screen')*. Whatever the user types is stored in *someVariable*.
- Key Idea 2: The *disp* function displays its input argument to the screen, and then automatically skips to the next line.
- Key Idea 3: The *fprintf* function displays its input argument to the screen, but does not automatically go to the next line. To do this, use the `\n` character inside the string to be displayed. You can quite literally use `\n` as a character– `\n\n\n` is the equivalent of hitting enter thrice in a word processor.
- Key Idea 4: You can insert values into an *fprintf* statement by using placeholders inside the character string that will be displayed. Use `%s` if the placeholder will be replaced by text (a character string) and `%f` if the placeholder will be replaced by a number.
- Key Idea 5: Following the display string, type the names of variables containing values (or values themselves) that should replace the placeholders, in the same order as the placeholders i.e. `fprintf('first the string %s and then the number %f \n', 'hello', 5)`.

- Key Idea 6: If you instead use the placeholder `%.2f` for numbers, the number will be rounded (as if using the *round* function), to 2 decimal places in this example.

8 Review- Vectors and Matrices

8.1 Creating Them

- Key Idea 1: To create a row vector, enclose a bunch of numbers in square brackets i.e. `x = [5 7 3]`.
- Key Idea 2: To create a matrix or column vector, use a semicolon within the square brackets to skip to the next line i.e. `x = [ 5 ; 7 ; 3 ]`.
- Key Idea 3: Use the colon operator from, say, *1 : 10* , to create a vector of points from 1 to 10, spaced by 1.
- Key Idea 4: Use the colon operator from, say, *1 : 0.02 : 10* , to create a vector of points from 1 to 10, spaced by 0.02.
- Key Idea 5: Use the *linspace* function, say with *linspace(1,5,100)*, to create a vector of 100 points evenly spaced between 1 and 5. Note that the colon operator specifies the spacing, and that's the **middle** number, whereas *linspace* specifies how many points are in the vector, and that's the **final** number.
- Key Idea 6: The function call *ones(5)* creates a 5x5 matrix containing 1 as each element, while *ones(5,3)* creates a 5x3 matrix containing 1 as each element. The functions *zeros* (each element is 0) and *eye* (for creating the identity matrix) work identically.
- Key Idea 7: Want a matrix containing all the same element, but something that's not 0 or 1? For instance, a 4x6 matrix containing only the number 5? Use *zeros(4,6)+5* or *ones(4,6)*5*.

8.2 Accessing Vectors/Matrices

- Key Idea 1: Use a single number. For row and column vectors stored in the variable *x*, *x(5)* gives you the fifth element. This approach also works for a matrix; the elements are numbered first going down the first column, then going down the second, and so on.
- Key Idea 2: Specify Row,Column. For matrices stored in the variable *x*, *x(3,5)* gives you the element in the third row and fifth column.
- Key Idea 3: Rows come first, followed by columns, when specifying a two number location in a matrix.
- Key Idea 4: *x = M(5)* saves a copy of the fifth element of the matrix *M* to the variable *x*. *M(5) = 12* sets the fifth element of the matrix *M* to be 12.
- Key Idea 5: Use the colon operator to specify multiple elements. The statement *M(1:3,2:5)* specifies the elements of matrix *M* that are in Rows 1 through 3 AND columns 2 through 5. Notice that this will give you a 3x4 matrix.

- Key Idea 6: Instead of using the colon operator, just use a vector. (This is the general case of the previous idea). i.e. *M(4,[1 4 6])* is equivalent to *[M(4,1) M(4,4) M(4,6)]*.
- Key Idea 7: A colon by itself means "all of the..." or "every". For instance, *M(5,:)* gives you only the fifth row of *M*, including the elements in every column.
- Key Idea 8: The keyword *end* means "the last". *M(5,4:end)* specifies the elements of the matrix *M* in the fifth row and all of the columns from the fourth through the last.
- Key Idea 9: Turn a matrix into a vector by typing something like *M(:)* (literally- give me all the elements of M).

8.3 Vector/Matrix Functions

- Key Idea 1: *size(M)* returns a 2 element vector containing the number of rows and number of columns in matrix *M*. Even better, call this as *[r c] = size(M)*. You often use *size* with matrices.
- Key Idea 2: *length(M)* returns only the larger dimension of *M*. For instance, if *M* were either 1x5 or 5x1, *length(M)* would return 5. You often use *length* with vectors.
- Key Idea 3: Functions on matrices generally work on each column individually. These functions include *sum, prod, max, min, sort, mean, and median*.
- Key Idea 4: To find the overall sum, product, etc., use the relevant function twice. i.e. *sum(sum(M))*. This works because the innermost *sum(M)* gives you a vector of the sums in each column, and the next *sum* is thus just summing a vector. Note that this method doesn't work for *median*. To successfully find the median, or as an alternative to calling the function twice, first turn the matrix into a vector i.e. *median(M(:))*.
- Key Idea 5: If you request two outputs from *max* or *min*, the first variable will contain the value of the maximum/minimum, and the second variable will contain the location in the vector. You can call this as *[a b] = max(M)*.
- Key Idea 6: The function *sortrows* alphabetizes a matrix.
- Key Idea 7: When X contains a matrix or vector, a statement like $X > 5$ returns a matrix or vector of the same size containing 1s or 0s (trues or falses) for each element. Namely, this tells you for each element if the condition is true or false.
- Key Idea 8: You can use the matrix of trues and falses to index a vector or matrix. Let's say that M is a matrix. The statement `M(M>100 & M<200)` returns a **vector** of all elements in that matrix that are between 100 and 200. However, `sum(M>100 & M<200)` returns a count of the number of elements between 100 and 200 since it is summing the logical vector (which is 1 every time the condition is true for an element).
- Key Idea 9: The *find* function returns a vector (indexing with the single element method) of all of the locations where a condition is true. For instance to find all locations in the matrix M that contain the largest element in the matrix, use *find(M = = max(max(M)))*. As in the previous idea, you can use this vector of locations to index the matrix *M*.

- Key Idea 10: The *transpose* function (or apostrophe) swaps the rows and the columns of a matrix. With a vector, notice that this will turn a row vector into a column vector, and vice versa.

- Key Idea 11: [x y] = meshgrid(a,b) returns two 'length(b)' row by 'length(a)' column matrices x and y, with gradients going horizontally and vertically, respectively. Note that `x.*y` gives you every possible product of one element each from the 'a' and 'b' vectors.

8.4 Vector/Matrix Math

- Key Idea 1: If you have two vectors or matrices of equal size and want to add, subtract, multiply, divide, or exponentiate the corresponding elements (and get an answer of the same size as the operands), use the following operators: `+ , - , .* , ./ , and .^`

- Key Idea 2: The operators `* , / , and ^` perform matrix multiplication, division, and repeated multiplication, respectively. In this course, you don't use those much. In linear algebra and higher level engineering courses, you use those constantly.

9 Strings Are Vectors; We Need Cell Arrays

- Key Idea 1: A string is a vector of single characters. Thus, *['hi' 'bye']* gives you the single string *'hibye'*, not a vector of two individual strings.

- Key Idea 2: To store strings inside a "vector", use cell arrays. Instead of using square brackets to create them, use squigly braces i.e. `x = { 'hi' 'bye' }`. To access the elements as strings, also use squigly braces (in place of parentheses) i.e. `disp(x{3})`.

- Key Idea 3: You can't use $==$ to compare strings. Instead, use the function *strcmp*. For instance, *strcmp(suit,'black')* will return 1 is the variable *suit* contains the string "black", and 0 if the suit is NOT black... High five!

10 Polynomial Roots, Systems of Equations

- Key Idea 1: Remember that a polynomial can be represented as a vector of its coefficients? For instance, $5x^3 + 6x + 3$ can be written as *[5 0 6 3]*.

- Key Idea 2: Use the *roots* function to find the roots (also called zeros) of the polynomial. This function returns a vector of the values (both real and imaginary) of x for which y is 0. For instance, for the polynomial $y = 5x^3 + 6x + 3$, use *roots([5 0 6 3])*.

- Key Idea 3: Don't confuse *roots* and *zeros*, they are very different functions.

- Key Idea 4: A system of equations describes when you have N equations containing N variables. Make a matrix A of the coefficients of the equations. Each row should represent an equation, and each column should represent a particular variable. Then make a **column** vector of the what each equation equals (the constant). Then, use **left division**: `A \ B`. This gives you a vector of N elements; these are the values of the variables represented by each column, in order.

Given Example System 1:

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

```
A = [ 1 5 2 ; 2 5 2 ; -6 -2 -1 ];  % coefficients
B = [ 2 ; 4 ; -6 ];  % the right-hand side
A \ B
     ans =
      2.0000
     12.0000
    -30.0000
```

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