MATLAB Primer for CAAM 336

Contents

1 Introduction

MATLAB is an important tool for better understanding the analytical and numerical methods for solving and approximating the partial differential equations we study in CAAM 336. While CAAM 210 provides a

great introduction to the use of MATLAB for scientific computing, this set of notes provides a refresher of the most important programming concepts and techniques that are needed to be successful in CAAM 336. Fully understanding these concepts and techniques will allow the reader to be able to better focus on the material of the course without being bogged down or intimidated by the assignments that require MATLAB programming.

Of course, MATLAB is a powerful program and we can't address all of its capabilities here. The MATLAB command **help** is a great tool if you are ever unsure of how to use a MATLAB function. For example, we can see what the **linspace** function, which we will discuss shortly, does by typing

```
>> help linspace
LINSPACE Linearly spaced vector.
    LINSPACE(X1, X2) generates a row vector of 100 linearly
    equally spaced points between X1 and X2.

    LINSPACE(X1, X2, N) generates N points between X1 and X2.
    For N < 2, LINSPACE returns X2.

    Class support for inputs X1,X2:
       float: double, single

    See also logspace, :.

    Reference page in Help browser
       doc linspace
```

2 Basic MATLAB commands and Linear Algebra

In this section we will discuss basic MATLAB commands such as creating vectors and matrices, building diagonal and tridiagonal matrices, and transposing matrices. We will also discuss basic linear algebra in MATLAB such as solving linear systems, computing eigenvalues and eigenvectors of a matrix, and computing vector norms. Finally we'll discuss how to "vectorize" your code in order to make it more efficient in MATLAB.

Before we begin, one of the most important parts of programming in general is *commenting your code*. In MATLAB this done by using %. Anything written after, but on the same line as, % will not be evaluated. This allows you to write a short descriptive comment about the purpose of this line or section of code. We strongly encourage you to comment as much of your code as possible. We will ask you to turn in your code and well-commented code will always get more partial credit than code without comments. Now without further ado...

2.1 Vectors

Let's start by manually creating a vector with values between 0 and 1, spaced by 0.2.

```
>> x=[0 .2 .4 .6 .8 1] % row vector
```

```
x =
         0    0.2000    0.4000    0.6000    0.8000    1.0000
>> x=x' % ' transposes vectors/matrices, creates here a column vector
x =
         0
    0.2000
    0.4000
    0.6000
    0.8000
    1.0000
>> x=[0;.2;.4;.6;.8;1]; % column vector
x =
         0
    0.2000
    0.4000
    0.6000
    0.8000
    1.0000
```

Now in general we want to specify vectors in a more efficient way. There are two main ways to do this. First, we can use the **linspace** command. It takes in three arguments: beginning value, end value, and the length of the vector.

```
>> n=5;
>> x=linspace(0,1,n+1)
x =
         0    0.2000    0.4000    0.6000    0.8000    1.0000
>> delta=1/n
delta =
    0.2000
>> spacing=x(2)-x(1)
spacing =
    0.2000
```

It is important to remember that the spacing of **linspace(a,b,n)** is $(b-a)/(n-1)$. The other way is define a vector by typing $(a : \Delta : b)$ where a is the starting value, Δ is the spacing, and b is the ending value IF $b-a$ is divisible by Δ. Otherwise it's the closest value less than b spaced by Δ starting from a. Note that both of these commands create row vectors not column vectors.

```
>> delta=0.2;
>> x=(0:delta:1) % length will be floor(1/delta) + 1
x =
         0    0.2000    0.4000    0.6000    0.8000    1.0000
>> delta = 0.15;
>> x=(0:delta:1)
x =
         0    0.1500    0.3000    0.4500    0.6000    0.7500    0.9000
```

Note that the length will be $\lfloor \frac{b-a}{\Delta} \rfloor$. As an aside, let's remember what the functions **floor**, **ceil**, and **round** do.

```
>> x=1.4;
>> [round(x) ceil(x) floor(x)]
ans =
     1     2     1
>> x=-1.4;
>> [round(x) ceil(x) floor(x)]
ans =
    -1    -1    -2
```

2.2 Matrices

Next let's create matrices. The commands **zeros**(m, n) and **ones**(m, n) create $m \times n$ matrices of all zeros and all ones, respectively. If either **zeros** or **ones** is given only one argument, it creates a square matrix of that size. Now suppose we want a matrix that is $m \times n$ where each row is identical. We can do this using the following commands:

```
>> n=4;
>> m=3;
>> x=linspace(0,1,n) % produces a 1xn vector
>> y=ones(m,1); % produces a mx1 vector of all ones
>> Z=y*x % produces an mxn matrix where each row is x by computing the outer product
Z =
         0    0.2500    0.5000    0.7500    1.0000
         0    0.2500    0.5000    0.7500    1.0000
         0    0.2500    0.5000    0.7500    1.0000
```

How can you make a matrix where each column is the same?

Now suppose we want a diagonal matrix with the same value on the diagonal. This is equivalent to the diagonal value times the identity matrix, which in MATLAB can be created using the **eye** function:

```
>> a=2; % diagonal value
>> n=4; % size of diagonal matrix
>> A=a*eye(n)
A =
     2     0     0     0
     0     2     0     0
     0     0     2     0
     0     0     0     2
```

Now construct a diagonal matrix where the diagonal matrices can be different:

```
>> b=[2 3 4 5];
>> A=diag(b)
A =
     2     0     0     0
```

```
     0     3     0     0
     0     0     4     0
     0     0     0     5
```

diag(v) creates a diagonal matrix where the diagonal elements are specified by an input vector v. **diag**(v,k) can also be used to create matrices by specifying sub and super diagonal entries at diagonal k:

```
>> A=diag(b,1)
A =
     0     2     0     0     0
     0     0     3     0     0
     0     0     0     4     0
     0     0     0     0     5
     0     0     0     0     0
>> B=diag(b,-2)
B =
     0     0     0     0     0     0
     0     0     0     0     0     0
     2     0     0     0     0     0
     0     3     0     0     0     0
     0     0     4     0     0     0
     0     0     0     5     0     0
```

Note the size of the matrix will be $(length(b) + |k|) \times (length(b) + |k|)$. Thus in order to create a matrix with different diagonals we need to be careful with the sizes of the input vectors:

```
>> C=a*eye(5)+diag(b,1)+diag(b,-1)
C =
     2     2     0     0     0
     2     2     3     0     0
     0     3     2     4     0
     0     0     4     2     5
     0     0     0     5     2
```

A commonly used matrix in 336 will be a matrix with 2's on the diagonal and -1's on the sub and super diagonal. We can create this using the following code:

```
>> n=4;
>> A=2*eye(n)-diag(ones(n-1,1),1)-diag(ones(n-1,1),-1)
A =
     2    -1     0     0
    -1     2    -1     0
     0    -1     2    -1
     0     0    -1     2
```

While not necessarily useful for CAAM 336, it is nice to be able to create matrices with random entries. MATLAB has two common commands for computing random numbers: **rand**(m, n) and **randn**(m, n). The first produces an $m \times n$ matrix of numbers distributed uniformly at random between 0 and 1. The second

produces an $m \times n$ matrix of numbers with Gaussian distribution with mean 0 and variance 1. Again, both functions, if only given one input, produce a square matrix with that number of rows and columns:

```
>> A=rand(2,5)
A =
    0.9270    0.4140    0.1809    0.1300    0.3455
    0.5343    0.0524    0.4979    0.7508    0.0576
>> A=rand(4)
A =
    0.0621    0.3527    0.0347    0.5507
    0.2898    0.8446    0.8017    0.4166
    0.0712    0.1093    0.5941    0.4567
    0.6813    0.0190    0.0145    0.2339
>> B=randn(4,1)
B =
    0.7515
   -0.5108
   -0.3383
   -0.1920
>> B=randn(3)
B =
    0.4761    1.0347   -0.0082
   -1.5106   -0.5344   -0.7206
   -0.3596    0.5172   -0.0001
```

2.3 Other Useful Commands

It is good to be comfortable with the **max**, **min**, and **sum** functions in MATLAB:

```
>> A=rand(4,2)
A =
    0.5190    0.4930
    0.9046    0.3629
    0.7920    0.4021
    0.3936    0.7668
>> max(A) % This gives you the maximum of each column.
ans =
    0.9046    0.7668
>> max(A,[],1) %  gives you maximum of each column
ans =
    0.9046    0.7668
>> max(A,[],2) % gives you the maximum of each row
ans =
    0.5190
    0.9046
    0.7920
```

```
    0.7668
% min works the exact same way
>> sum(A) % This gives you the sum of each column of A.  ans =
    2.6093    2.0248
%   To specify which direction you want the sum:
>> sum(A,1) % also gives you sum of each column of A, where
ans =
    2.6093    2.0248
>> sum(A,2) % gives you the sum of each row of A
ans =
    1.0121
    1.2675
    1.1941
    1.1604
% A helpful way to remember which 1 and 2 refer to is to think that mxn
% matrices have m rows and n columns, so the first dimension is rows and
% second is columns, thus 1 will return a row and 2 will return a column;
% the same holds for max, min, and sum
>> n=5;
>> x=(1:n)
x =
     1     2     3     4     5
>> y=x.^2 % this squares each element in the matrix/vector, note the use of the "."
y =
     1     4     9    16    25
```

The "." is often necessary when you want to apply a scalar command element-wise to a vector or matrix. Other examples include element-wise multiplication and division:

```
>> y.*x  % gives the product of x and y elementwise
ans =
     1     8    27    64   125
>> y./x  % gives the division of y by x elementwise
ans =
     1     2     3     4     5
```

To use builtin matlab functions like **exp**, **sin**, **cos**, etc we can apply these element-wise without using the "."

```
>> exp(x) % returns the vector whose entries are e^x for each value of x
ans =
    2.7183    7.3891   20.0855   54.5982  148.4132
```

Now let's combine these to compute $xe^{-\sin(x)^2/2}$ for $x = 0, 1, 2, \ldots, 5$:

```
>> x=(0:5);
>> y=x.*exp(-(sin(x)).^2/2)
y =
         0    0.7018    1.3228    2.9703    3.0039    3.1572
```

2.4 Solving linear systems

Let's now solve a linear system: $Ax = b$ where A is an $n \times n$ matrix, b is a given $n \times 1$ vector and x is an unknown $n \times 1$ vector

```
>> A=randn(4)
A =
   -0.3435   -1.4655    0.5115   -0.4790
   -0.6014   -0.4059   -0.4129   -0.3083
    0.2484   -0.2613    1.3552   -0.3268
   -2.3640   -0.7164   -1.2159    0.2462
>> b=randn(4,1)
b =
   -0.9155
    0.3530
   -0.3335
   -0.4871
>> x=A\b  % the \ (backslash) command in matlab solves the linear system
x =

   -0.0674
    1.0678
   -0.4620
   -1.8008
% To verify:
>> b-A*x
ans =
   1.0e-15 *

    0.1110
         0
   -0.1110
    0.0555
```

Note here that MATLAB doesn't return exactly a vector of all 0's. This is due to the fact that MATLAB uses numerical methods to solve the linear system that involve using finite precision. 10^{-16} is MATLAB's "machine epsilon" and is essentially as accurate as you can hope to get your solution numerically. So when you see numbers around this size, just think zero! So up to machine precision, we have solved our linear system.

Another way to solve the linear system is to compute the inverse of the matrix A and multiply it times b:

```
>> x=inv(A)*b
x =
   -0.0674
    1.0678
```

```
   -0.4620
   -1.8008
>> b-A*x
ans =
   1.0e-15 *

    0.2220
         0
         0
         0
```

In general we want to avoid computing matrix inverses. One reason is because they take longer to compute because they require more numerical operations than solving a linear system by other methods. The other reasons are more complex and have to do with stability of finite precision numerical computations. In MATLAB, the computation time of a section of code can be computed by calling **tic** before the command and **toc** afterwards.

As an aside, there is a lot of mathematics behind how the backslash command in MATLAB solves linear systems efficiently and accurately. To learn more about the beautiful mathematics behind the algorithms to solve linear systems, consider taking CAAM 453, 454, 551.

2.5 Vector Norms

When we computed $b - Ax$, we were trying to quantify how accurate our solution to the linear system was. There are many different ways to quantify the size of this error. For example, if we define the "residual" $e = b - Ax$ then we could call the maximum absolute value of e as our error. The average value of e would be another possible choice of the error. Yet another choice could be the Euclidean length of the vector e. In mathematics we call these different choices of size "metrics" and certain types of metrics are called "norms". In CAAM 336 we will often try to approximate solutions and then ask you to quantify the error of the approximation using different metrics and norms.

Let us calculate a few different error metrics: First let's construct a random vector and a random error vector:

```
>> n=4;
>> x=randn(n,1);
>> y=x+.01*randn(n,1);
>> e=y-x
e =
    0.0036
    0.0020
    0.0031
   -0.0068
>> m1=max(abs(e)) % This is the maximum absolute value of the error vector e
m1 =
    0.0068
```

```
>> m1b=norm(e,inf)  % This is an equivalent way to compute the maximum absolute value of a vector.
% It is called the infinity norm.
m1b =
    0.0068
>> m2=mean(e)   % This is the average value of e
m2 =
   4.7841e-04
>> m3=norm(e)   % This is the Euclidean lengh of e, equivalently we can compute it:
m3 =
    0.0085
>> m3b=sqrt(sum(e.^2))  % Note that m3 and m3b are the same
m3b =
    0.0085
>> m4=norm(e,1)  % This is the sum of the absolute values of e.  It is called the 1-norm
m4 =
    0.0155
>> m4b=sum(abs(e))  % Note m4 and m4b are the same
m4b =
    0.0155
```

2.6 Eigenvalues and Eigenvectors

Eigenvalues and eigenvectors are fundamental to linear algebra and mathematics in general as you will learn in CAAM 336. As a reminder, we call λ an eigenvalue and v an eigenvector if they satisfy the equation:

$$Av = \lambda v \qquad v \in \mathbb{R}^n, \lambda \in \mathbb{R}$$

for a square matrix $A \in \mathbb{R}^{n \times n}$. In CAAM 336, we extend the concept of eigenvalues and eigenvectors to other linear operators A that might not be matrices, but for now we will stick to matrices. The following is a demo of the **eig** function in MATLAB

```
>> A = [1 1; 0 2]
A =
     1     1
     0     2
>> lambda = eig(A)     % with one argument, eig returns only the eigenvalues
lambda =
     1
     2
>> [V,D] = eig(A)      % with two arguments, eig returns matrices V and D
                       % with eigenvectors as the columns of V and
                       % eigenvalues on the diagonal entries of D.
V =
    1.0000    0.7071
         0    0.7071
D =
```

```
     1     0
     0     2
>> [A*V V*D]           % note that A*V = V*D
ans =
    1.0000    1.4142    1.0000    1.4142
         0    1.4142         0    1.4142
>> A*V-V*D
ans =
     0     0
     0     0
>> v1 = V(:,1)         % first column of V
v1 =
     1
     0
>> v2 = V(:,2)         % second column of V
v2 =
    0.7071
    0.7071
>> lam1 = D(1,1)       % first eigenvalue
lam1 =
     1
>> lam2 = D(2,2)       % second eigenvalue
lam2 =
     2
>> [A*v1 lam1*v1]
ans =
     1     1
     0     0
>> A*v1 - lam1*v1      % indeed, lam1 and v1 are an eigenvalue/eigenvector pair
ans =
     0
     0
>> [A*v2 lam2*v2]
ans =
    1.4142    1.4142
    1.4142    1.4142
>> A*v2 - lam2*v2      % indeed, lam2 and v2 are an eigenvalue/eigenvector pair
ans =
     0
     0
>> A = [1 1; -1 1]     % new matrix
A =
     1     1
    -1     1
>> [V,D] = eig(A)      % compute eigenvalues/eigenvectors
```

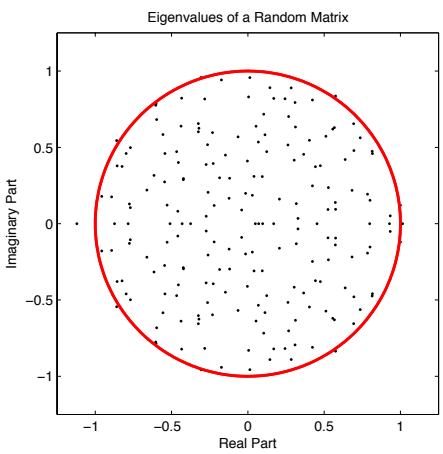
```
V =
   0.7071             0.7071
        0 + 0.7071i        0 - 0.7071i
D =
   1.0000 + 1.0000i        0
        0             1.0000 - 1.0000i
% A has real entries.... but complex eigenvalues and eigenvectors
>> v1 = V(:,1); v2 = V(:,2);
>> lam1 = D(1,1); lam2 = D(2,2);
>> [A*v1 lam1*v1]
ans =
   0.7071 + 0.7071i   0.7071 + 0.7071i
  -0.7071 + 0.7071i  -0.7071 + 0.7071i
>> A*v1 - lam1*v1
ans =
     0
     0
>> [A*v2 lam2*v2]
ans =
   0.7071 - 0.7071i   0.7071 - 0.7071i
  -0.7071 - 0.7071i  -0.7071 - 0.7071i
>> A*v2 - lam2*v2
ans =
     0
     0
```

In fact, "typical" matrices with real entries have complex eigenvalues. Next we look at an example of a random matrix.

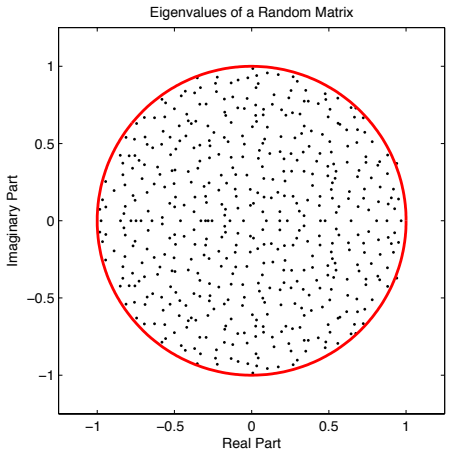
```
>> A = randn(100)/sqrt(100);   % entries normally distributed, mean 0, variance 1/100
>> lam = eig(A);
>> plot(real(lam),imag(lam),'k.')
>> axis equal, axis([-1.25 1.25 -1.25 1.25])
```

As the matrix dimension gets large, these eigenvalues tend to cover the unit circle in the complex plane with uniform probability. To see this, we plot the unit circle...

```
>> T = exp(linspace(0,2i*pi,500));
>> hold on, plot(real(T),imag(T),'r-','linewidth',2)
% repeat this experiment for a larger matrix
>> A = randn(500)/sqrt(500);
>> lam = eig(A);
>> figure
>> plot(real(lam),imag(lam),'k.')
>> axis equal, axis([-1.25 1.25 -1.25 1.25])
>> hold on, plot(real(T),imag(T),'r-','linewidth',2)
```



(a) Eigenvalues of $A \in \mathbb{R}^{100\times 100}$



(b) Eigenvalues of $A \in \mathbb{R}^{500\times 500}$

In fact, these eigenvalues tend to cover the unit circle "more uniformly" than at random. Let's do a side-by-side plot using eigenvalues on one side and random numbers on the other.

```
>> coord=2*rand(1000,2)-1;
>> normCoord=sqrt(sum(coord.^2,2));
>> coord2=coord(normCoord<=1,:);
>> coord=coord2(1:500,:);
```

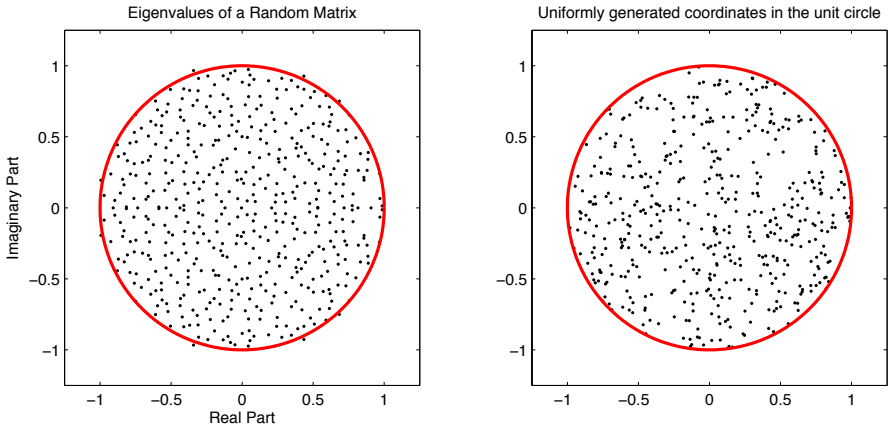
coord now represents 500 coordinate pairs that fall within the unit circle. This is not the most efficient implementation because we had to throw away some of our randomly generated values. (Can you think of another way?) Polar coordinates will work but you need to be careful because uniformly choosing the radius will bias the values towards the center of the circle.

```
>> figure;
>> subplot(1,2,1);
>> plot(real(lam),imag(lam),'k.')
>> axis equal, axis([-1.25 1.25 -1.25 1.25])
>> hold on, plot(real(T),imag(T),'r-','linewidth',2)
>> title('Eigenvalues of a Random Matrix');
>> subplot(1,2,2);
>> plot(coord(:,1),coord(:,2),'k.');
>> axis equal, axis([-1.25 1.25 -1.25 1.25])
>> hold on, plot(real(T),imag(T),'r-','linewidth',2)
>> title('Uniformly generated coordinates in the unit circle');
```

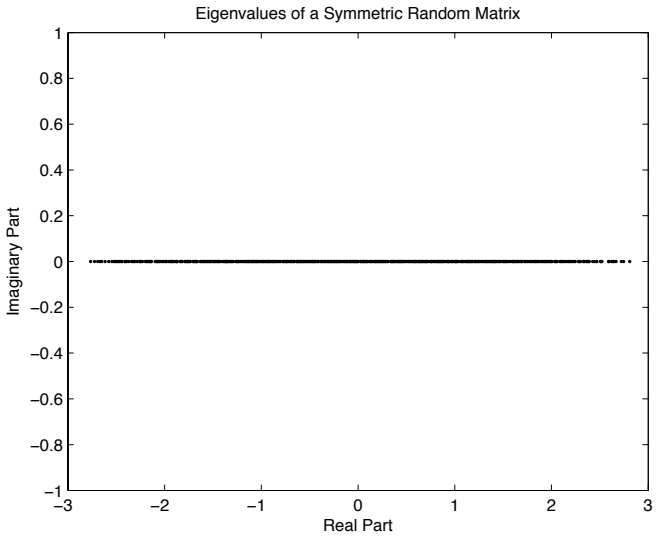
This is quite amazing. It appears that the eigenvalues "repel" each other so that they are more spread out than simply being uniformly random.

Random SYMMETRIC matrices act very differently. Here's one example.

```
>> Asym = A + A';
>> norm(Asym-Asym')    % If Asym = Asym', then this norm should be zero
```




```
ans =
     0
>> lam = eig(Asym);
>> figure
>> plot(real(lam),imag(lam),'k.')
```



Now all the eigenvalues look to be real numbers! Let's check this:

```
>> max(abs(imag(lam)))
ans =
     0
```

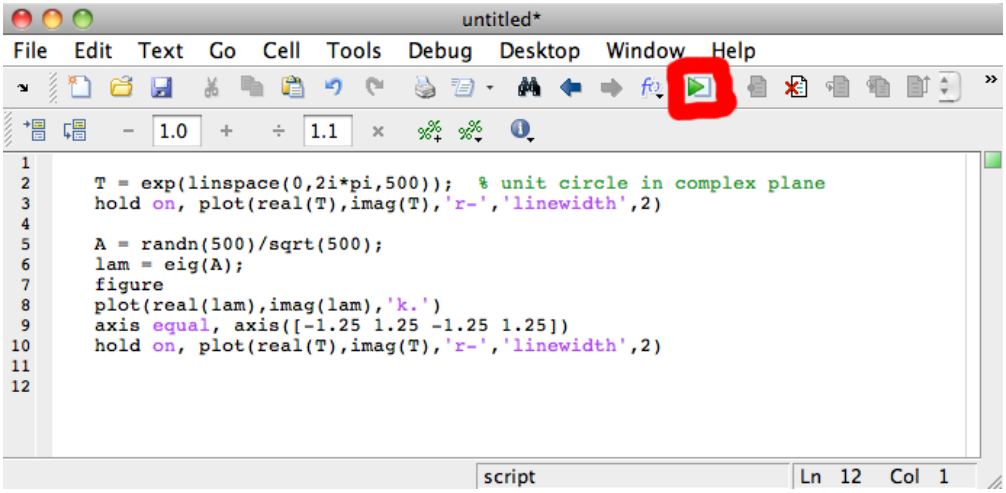
This is a remarkable property: real symmetric matrices ALWAYS have real eigenvalues. We will prove this later this semester in CAAM 336.

3 Defining Functions

3.1 Functions, Scripts, and Cells

There are three main ways to use MATLAB: on the command line, in a script, or in a function. Thus far we have been giving examples while using the command line. However, for all but the most simple tasks, you should create a script or a function that stores your commands. It is our philosophy that you should primarily use scripts, while functions should be used only when you have a small piece of code with simple inputs that you will call multiple times. But before we get ahead of ourselves, let's describe what is the difference between scripts and functions in MATLAB.

In MATLAB, if you go to File, New, and then choose Script, it will open a blank editor. In this editor, you can simply enter various commands one line at a time. By ending a line with a semicolon, that line's output will be suppressed. By default, you should end each line with a semicolon. Here is an example script of the code to generate and plot eigenvalues of random matrices we just discussed. To execute this script

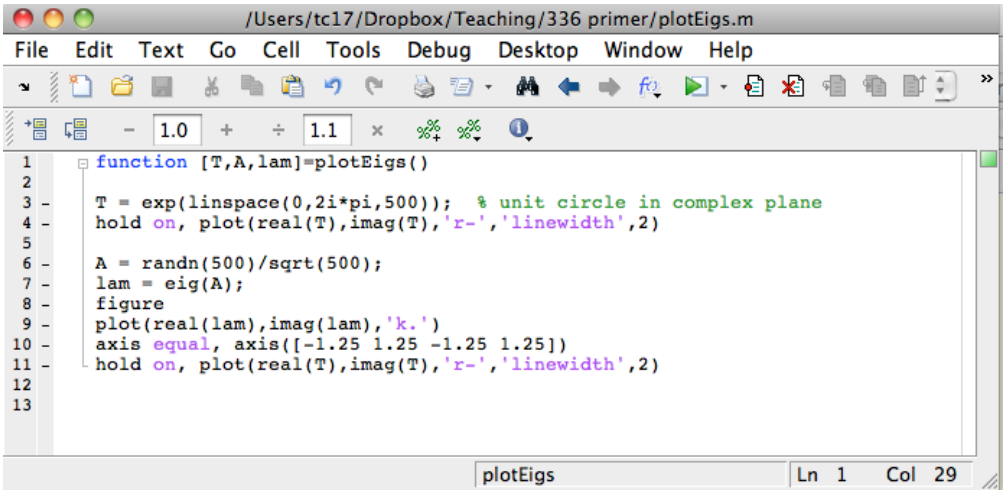


you simply need to save it and click on the button that looks like a green play button in the toolbar (circled in red in the figure).

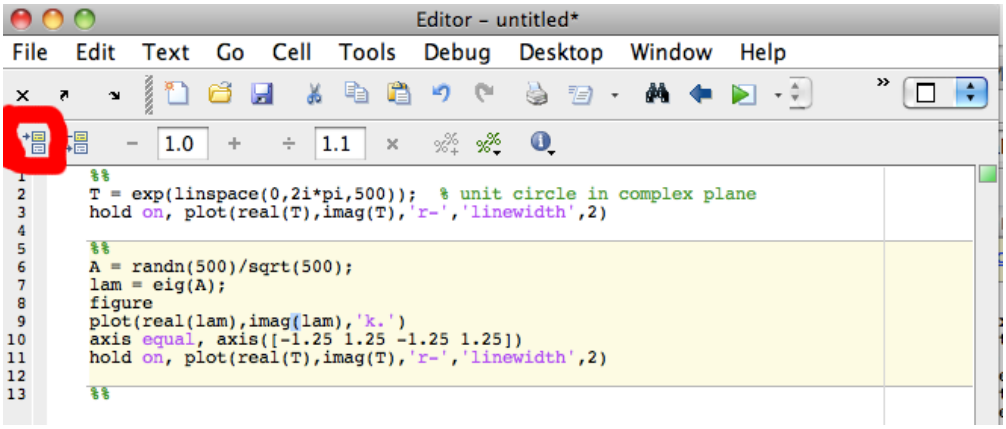
To contrast, functions have a more rigid structure. They must start with a statement of the following form:

```
function [return1 return2]=functionTitle(input1,input2,input3)
```

where return1 and return2 are output values and input1, input2, and input3 are input values to the function. A function can have as many or few input and return values as you desire. Here is an example of a function To execute this function, you should call it from the command prompt or from another script or function. One negative about functions is that you do not have access to the local variables inside the function, only the output values. This is a BIG deal when you are editing your code and want to test and debug it. Note: since this function has no inputs, it could be run by pressing the execute button as before. However, you will not have access afterwards to the outputs because you are not actually storing them at the command prompt.



Finally, within scripts (and functions), you can define cells by separating sections of the code with a line that begins with two percent signs (%%). This feature is great when you have multiple different sections in your script and you want to execute part of it without executing all of it. In MATLAB, the cell you are currently in will have a yellowish background color. To evaluate the individual cell, you can click on the top left button on the toolbar right above your code (circled in red in the example figure below).



3.2 For Loops

For loops are a basic and essential tool in the programmers toolbox and it is one you will use often in the class. The main idea of a for loop is that it is a construction that allows for a block of code to be executed repeatedly over a specified index set. The MATLAB notation for the for loops is

```
n=100;
x=linspace(0,1,n);
for k=1:n
    f(k)=sin(pi*x(k));
```

```
end
```

This will evaluate the function $\sin(\pi x)$ at each of the points $x(k)$ for $k = 1, \ldots, n$ and store them into the vector f.

3.3 Vectorizing Code

Whenever it is possible, you should try to use built-in matlab functions instead of writing loops. This will make your code much faster. Here are some examples. We will use the commands **tic** and **toc** to time how long each takes:

Example 1: Building up a tridiagonal matrix of 2's on the main diagonal and -1 on the sub and super diagonal:

```
n=50;
A=zeros(5);
% double for loop:
tic
for i=1:n
    for j=1:n
        if(i==j)
            A(i,j)=2;
        elseif(abs(i-j)==1)
            A(i,j)=-1;
        end
    end
end
toc
% Note the above code would be even slower if we added an "else A(i,j)=0;"

% Using built-in MATLAB commands
tic
A=2*eye(n)-diag(ones(n-1,1),1)-diag(ones(n-1,1),-1);
toc
```

The times returned will vary based on your computer. On my laptop, MATLAB returned:

```
Elapsed time is 0.005800 seconds.
Elapsed time is 0.000109 seconds.
```

Example 2: Building up a matrix whose each column is the same:

```
n=50;
x=rand(n,1);
A=zeros(n);
tic
for i=1:n
    A(:,i)=x;
end
toc

tic
A=x*ones(1,n);
toc
```

MATLAB returned:

```
Elapsed time is 0.000079 seconds.
Elapsed time is 0.000036 seconds.
```

Example 3: Multiplying two matrices together elementwise

```
A=rand(n);
B=rand(n);
tic
for i=1:n
    for j=1:n
        C(i,j)=A(i,j)*B(i,j);
    end
end
toc

tic
C=A.*B;
toc
```

MATLAB returned:

```
Elapsed time is 0.006616 seconds.
Elapsed time is 0.000025 seconds.
```

Now for the most dramatic example. Let's compare computing matrix-matrix multiplication using the simplest possible implementation with the MATLAB implementation:

```
A=rand(n);
B=rand(n);
C=zeros(n);
tic
for i=1:n
    for j=1:n
        for k=1:n
            C(i,j)=C(i,j)+A(i,k)*B(k,j);
        end
    end
end
toc

tic
C=A*B;
toc
```

MATLAB returned:

```
Elapsed time is 0.387614 seconds.
Elapsed time is 0.000264 seconds.
```

If we increase n from 50 to 500 and repeat the above code, MATLAB returns:

```
Elapsed time is 349.491953 seconds.
Elapsed time is 0.232229 seconds.
```

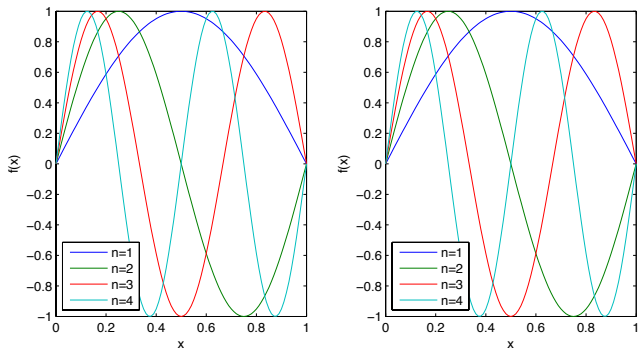
MATLAB's built-in approach is 1400 times faster for this example! Hopefully, you are now convinced that vectorizing your code and avoiding for loops is much faster in MATLAB.

3.4 Inline Functions

Often it is useful to define a function that has variable inputs. For example, suppose you want to use the function $f(x, n) = \sin(n\pi x)$ for various values of n and x. One way to do this is to use inline functions: using "@" or the command **inline**. Below we give examples of each method:

```
f=@(x,n) sin(n*pi*x);
g = inline('sin(n*pi*x)', 'x', 'n');
x=(0:.01:1);
n=(1:4)';
figure;
subplot(1,2,1); plot(x,f(x,n)'); xlabel('x'); ylabel('f(x)');
legend('n=1','n=2','n=3','n=4','Location','SouthWest');
```

```
subplot(1,2,2); plot(x,g(x,n)'); xlabel('x'); ylabel('f(x)');
legend('n=1','n=2','n=3','n=4','Location','SouthWest');
```



Note the use of transposes. For f (and g) to take two vectors as inputs, we needed the dimensions to make sense. With x being $1 \times N$ and n being $M \times 1$ after the transpose, when we compute $n\pi x$ we have dimensional agreement for matrix multiplication $(M \times 1) \times (1 \times N) = (M \times N)$. We transposed $f(x, n)$ (and $g(x, n)$) again because we wanted to plot the result over the x values for each fixed value of n.

3.5 Functions

For more complicated functions, it's better to define a new function within MATLAB. For example, suppose we wanted to define a piece-wise step function of variable step size and length. Let h be the constant step lengths and a be the vector of step sizes and x be the domain of the function. In our script we could write

```
h=.2;  %step size
a=randn(10,1); % step heights
x=linspace(0,1,1000); % discretization of domain
fxn = stepfxn(h,a,x);
figure; plot(x,fxn)
xlabel('x','Fontsize',16); ylabel('f(x)','Fontsize',16);
title('Step Function','Fontsize',16);
```

while we separately define the function **stepfxn** as

```
function fxn = stepfxn(h,a,x)
%
% Creates step function values at points x for a step
% function with step size h and step height a.
%
```

```
% Usage:  fxn = stepfxn(h,a,x)
%
% Inputs:
%
% h       - step size
%
% a       - step height
%
% x       - points to evaluate function
%
% Outputs:
%
% fxn     - function values at x

n=length(a);
m=length(x);
fxn=zeros(m,1);

acum=cumsum(a);

for j=1:n
    abc=find((x>(j-1)*h).*(x<=j*h));
    fxn(abc)=acum(j);
end
```

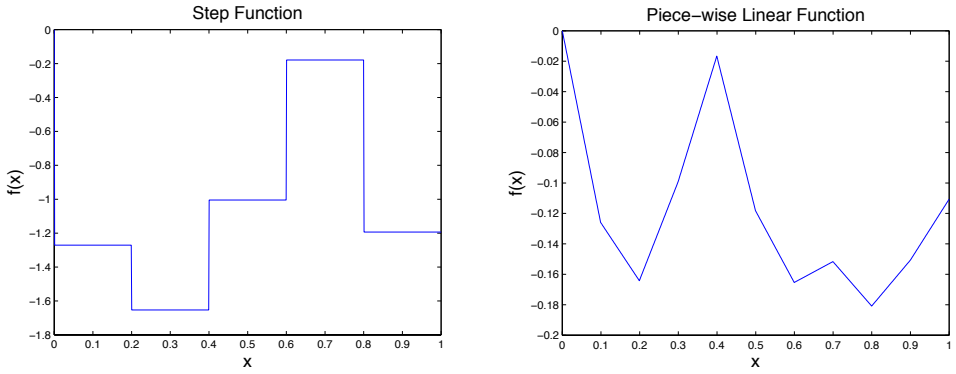


Figure 1:

As an aside, note that if we were to type **help stepfxn** we would see the header comments of the function:

```
>> help stepfxn
Creates step function values at points x for a step
  function with step size h and step height a.

  Usage:  fxn = stepfxn(h,a,x)
```

```
Inputs:

h       - step size

a       - step height

x       - points to evaluate function

Outputs:

fxn     - function values at x
```

Here another example is piece-wise linear function. First the script:

```
a=randn(10,1); % step heights
x=linspace(0,1,1000); % discretization of domain
fxn = pwlinearfxn(a,x);
figure; plot(x,fxn);
xlabel('x','Fontsize',16); ylabel('f(x)','Fontsize',16);
title('Piece-wise Linear Function','Fontsize',16);
```

The code for function **pwlinearfxn** is

```
function fxn = pwlinearfxn(a,x)
%
% Evaluates piece-wise linear function at values specified at points x in
% [0,1]
%
% Usage:  fxn = pwlinearfxn(a,x)
%
% Inputs:
%
% a       - vector of slopes, length corresponds to number of slope changes
%           in interval [0,1] uniformly spaced
%
% x       - points to evaluate function
%
% Outputs:
%
% fxn     - function values at x

n=length(a);
kinks=(1:n)/n;
m=length(x);
```

```
for j=1:m
    temp=find(x(j)>kinks);
    if isempty(temp)
        fxn(j)=a(1)*x(j);
    else
        behindKink=temp(end);
        fxn(j)=mean(a(1:behindKink))*kinks(behindKink)+a(behindKink+1)*(x(j)-kinks(behindKink));
    end
end
```

One realization of these scripts (because the step sizes are generated randomly) gives us the plots in Figure 1.

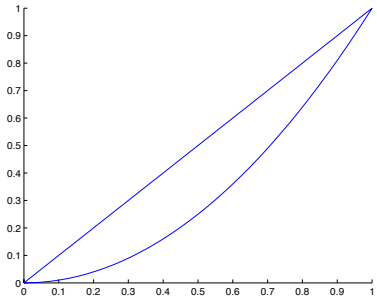
4 Tips for Making Figures

It is important to display your quantitative results in a clear manner. This section will outline the basics of plotting and emphasize how to label your plots in a quick, clear, and concise way.

4.1 Plotting Basics

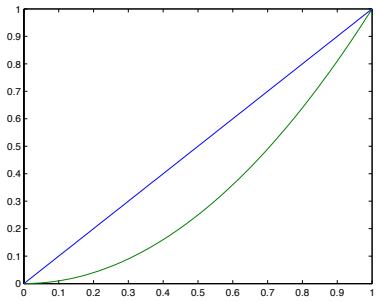
As a motivating problem, let's consider plotting the polynomials $f_n(x) = x^n$ for $n = 0, 1, \ldots$ on the interval $x \in [0, 1]$. First let's start with x and x^2. Consider the code below.

```
n=100;
delta=1/(n-1);
x=0:delta:1;
figure; % Opens up a new figure window
hold on;  % Keeps subsequent plots in the same figure
plot(x,x);  % Plots f(x)=x at the values chosen in vector x
plot(x,x.^2); % Plots f(x)=x.^2 on the same plot
```



It's nicer to have them in different colors. If we instead do this:

```
figure; plot(x,x,x,x.^2);
```

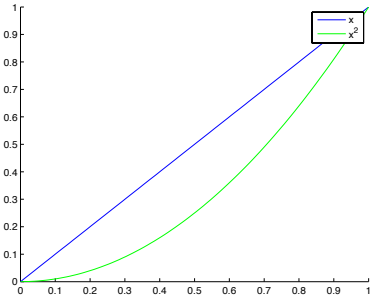


then we will get different colors. However sometimes we may want to plot these one at a time. How to do this an also properly label a plot is discussed next.

4.2 Labels, Legends, and Latex

Often you might have multiple plots that you want on the same figure but you don't want to plot them all in the same plot command. One way to get each to have a different color is to do the following. First, let's define a character string of colors in matlab: b: blue, g: green, r: red, c: cyan, m: magenta, y: yellow, k: black. We can use this as follows:

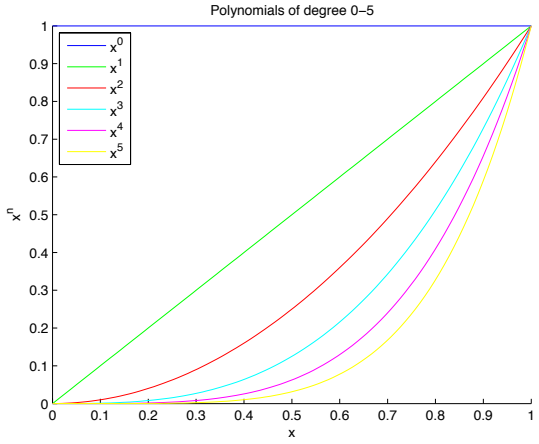
```
colors='bgrcmyk';
figure; hold on;
plot(x,x,colors(1));
plot(x,x.^2,colors(2));
legend('x','x^2');
```



Notice with multiple plots on the same figure, it is appropriate to have a legend to distinguish between them. This is done with the legend command.

Now let's combine what we've learned so far to plot in a loop the polynomials of degree 0, 1, 2, 3, 4, 5 on the same figure.

```
powers=0:5; % vector of polynomial powers
figure; hold on;
for j=1:length(powers)
    plot(x,x.^powers(j),colors(j))
    legendStr{j}=[‘x^’ num2str(powers(j))];
end
legend(legendStr,‘Location’,‘NorthWest’);
xlabel(‘x’);
ylabel(‘x^n’);
title(‘Polynomials of degree 0-5’);
```



Several things to note here:

1. By using a cell of strings (legendStr$\{j\}$ defines a cell of strings indexed by j) we can easily use the legend command on all of the plots simultaneously.

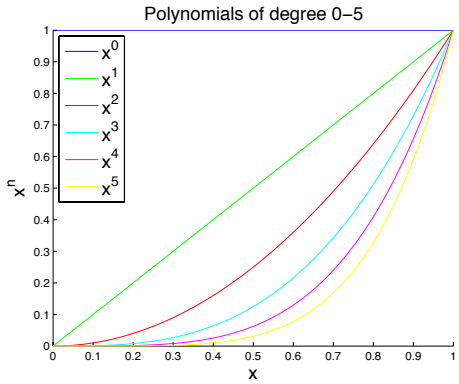
2. We can create strings that have characters and variable numbers in them by appending them using brackets and the command num2str as shown above.

3. The default location for the legend is the northeast position on the plot, but we can move it from covering up part of the plots by specifying its location.

4. To complete this plot, let us add labels for the x and y axis and also a title to our plot

One thing that is not satisfactory about these plots are that the legend, labels, and title are too small. We can fix this by specifying their sizes:

```
powers=0:5; % vector of polynomial powers
figure; hold on;
for j=1:length(powers)
    plot(x,x.^powers(j),colors(j))
    legendStr{j}=['x^' num2str(powers(j))];
end
legend(legendStr,'Location','NorthWest','FontSize',16);
xlabel('x','FontSize',16);
ylabel('x^n','FontSize',16);
title('Polynomials of degree 0-5','FontSize',16);
```



You can also specify the tick values and sizes on the x and y axis, but let's move on to something more useful. Suppose you wanted to label your figure with greek letters or other mathematical symbols. Using the latex interpreter, this can be done easily. In the example below we demonstrate this, specifying the line width of the plot, and the use of **subplot**. The resulting plot is in Figure 2.

```
theta=linspace(0,2*pi,100);
complexExp=exp(1i*theta);
figure;
subplot(1,2,1);
plot(theta,real(complexExp),'linewidth',4);
xlabel('$\theta$','interpreter','latex','fontsize',16);
ylabel('Re$\left(e^{i\theta}\right)$','interpreter','latex','fontsize',16);
title('Real Part of Complex Exponential','fontsize',16);
subplot(1,2,2);
plot(theta,imag(complexExp),'linewidth',4);
xlabel('$\theta$','interpreter','latex','fontsize',16);
ylabel('Im$\left(e^{i\theta}\right)$','interpreter','latex','fontsize',16);
title('Imaginary Part of Complex Exponential','fontsize',16);
```

Now let us do another example combining many of the tools we've learned thus far. Using the Taylor series expansion, let us approximate the exponential function $f(x) = e^x$ by polynomials. The Taylor series representation of a function $f(x)$ around $x = 0$ is:

$$f(x) = f(0) + f'(0)x + f''(0)\frac{x^2}{2!} + f'''(0)\frac{x^3}{3!} + ...$$

On the same plot let us compare the approximation of the Taylor series for different degree polynomials

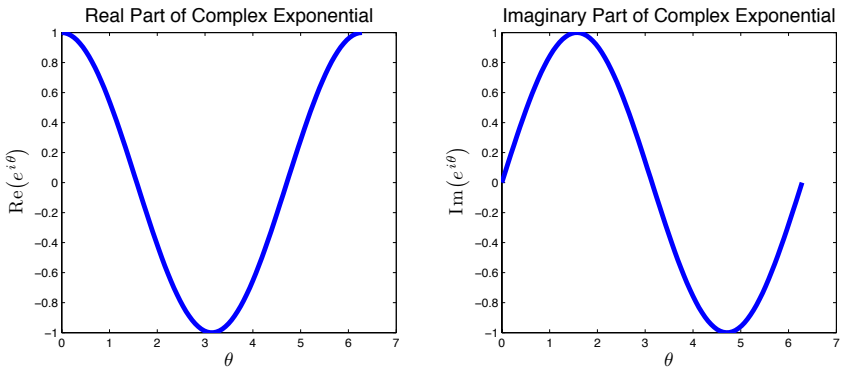
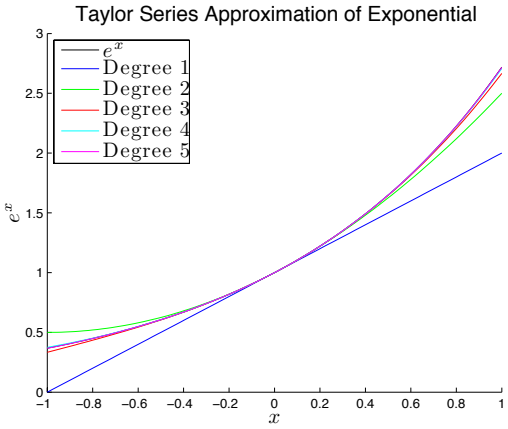


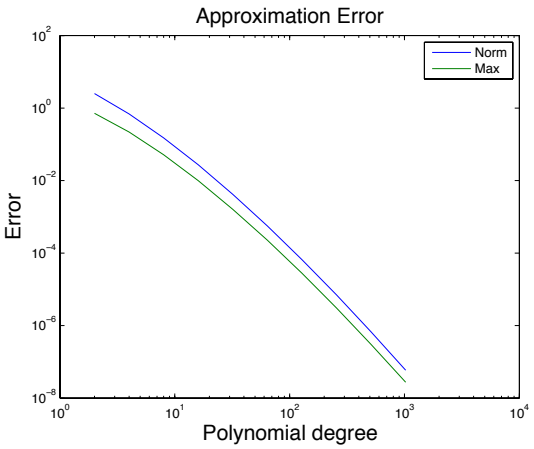
Figure 2:

```
x=linspace(-1,1,100);
y=exp(x);
degrees=1:5;
figure; hold on;
plot(x,y,'k');
approx=exp(0)*ones(size(x));
legendStr{1}='$e^x$';
for j=1:length(degrees)
    approx=approx+exp(0)*x.^j/factorial(j);
    plot(x,approx,colors(j));
    legendStr{j+1}=['Degree ' num2str(j)];
end
legend(legendStr,'Location','NorthWest',...
'Fontsize',16,'interpreter','latex');
title('Taylor Series Approximation of ...
Exponential','Fontsize',16);
xlabel('$x$','interpreter','latex','Fontsize',16);
ylabel('$e^x$','interpreter','latex','Fontsize',16);
```



Visually it looks like the approximation is getting better as we use more terms. To observe this more clearly, let's plot the error as we continue to increase the number of terms.

```
degrees=2.^(1:10);
figure;
approx=exp(0)*ones(size(x));
for j=1:length(degrees)
    approx=approx+exp(0)*x.^j/factorial(j);
    err(j)=norm(exp(x)-approx);
    err2(j)=max(abs(exp(x)-approx));
end
loglog(degrees,err,degrees,err2);
xlabel('Polynomial degree','Fontsize',16);
ylabel('Error','Fontsize',16);
legend('Norm','Max','Location','NorthEast');
title('Approximation Error','Fontsize',16);
```



4.3 Visualizing 3-D data

CAAM 336 is a class on partial differential equations and thus we will be dealing with equations involving functions of multiple variables. In order to display the values of a function of two variables, we need three dimensions: one for each variable and then one for the function value. In this subsection, we will address how we can do this in MATLAB. Let us first define the following function

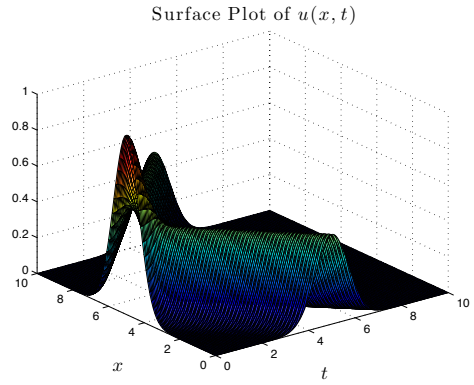
$$u(x,t) = \frac{1}{2}\left(e^{-(x-\ell/2-t)^2} + e^{-(x-\ell/2+t)^2}\right), \qquad x \in (-\infty,\infty), t \geq 0$$

If we interpret x to be a spatial variable and t to represent time, this function can be thought of as a traveling wave on an unbounded domain, but let us consider it only for $x \in [0,\ell]$ and $t \in [0,\ell]$. To visualize this function in MATLAB let us compute it's values at discrete points in x and t and store it in a matrix U. Then we can plot this matrix in several different ways as shown below.

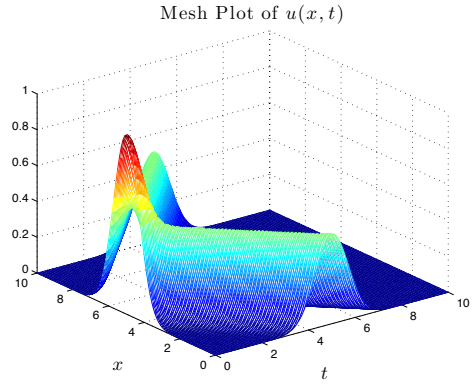
```
ell=10;
u=@(x,t) 0.5*(exp(-(x-ell/2-t).^2)+exp(-(x-ell/2+t).^2));
x=linspace(0,10,101);
t=linspace(0,10,200);
for i=1:length(t)
    U(:,i)=u(x,t(i));
end
```

4.3.1 3-D commands: surf, mesh, waterfall, imagesc

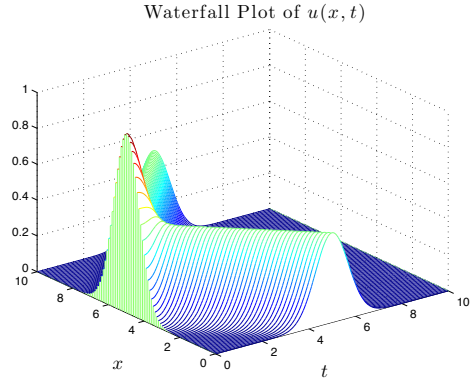
```
%% Using surf
figure;
surf(t,x,U);
xlabel('$t$','interpreter','latex','Fontsize',16);
ylabel('$x$','interpreter','latex','Fontsize',16);
title('Surface Plot of $u(x,t)$','interpreter',...
'latex','Fontsize',16);
```



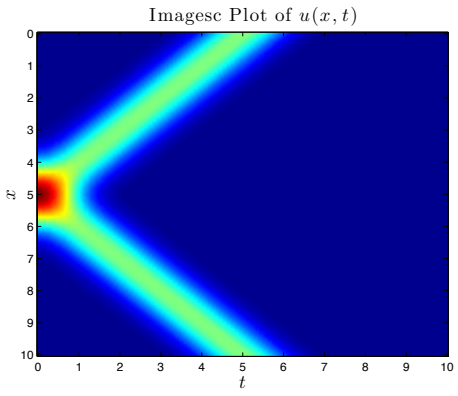

```
%% Using mesh
figure;
mesh(t,x,U);
xlabel('$t$','interpreter','latex','Fontsize',16);
ylabel('$x$','interpreter','latex','Fontsize',16);
title('Mesh Plot of $u(x,t)$','interpreter',...
'latex','Fontsize',16);
```




```
%% Using waterfall
figure;
waterfall(t,x,U);
xlabel('$t$','interpreter','latex','Fontsize',16);
ylabel('$x$','interpreter','latex','Fontsize',16);
title('Waterfall Plot of $u(x,t)$','interpreter',...
'latex','Fontsize',16);
```




```
%% Using imagesc
figure;
imagesc(t,x,U);
xlabel('$t$','interpreter','latex','Fontsize',16);
ylabel('$x$','interpreter','latex','Fontsize',16);
title('Surface Plot of $u(x,t)$','interpreter',...
'latex','Fontsize',16);
```



Each of the plots made using **surf, mesh, waterfall** can be rotated to be viewed from different perspectives. This can be done manually in the GUI or in the code using the command **view**.

4.3.2 Using pause and making movies

When visualizing a solution that is a function of time, it is often more instructive to plot the function sequentially for each time value to see how it changes in time. If you do this in a for loop in MATLAB, it will plot so fast that you only see the final result. To get around this, we can use the **pause** command. See the example code below for $u(x,t)$ plotted over x for each time t that we computed above and stored in U.

```
figure;
for i=1:length(t)
    plot(x,U(:,i));
    axis([0 10 0 1.1]);
    title(['t=' num2str(t(i))],'interpreter','latex');
    pause(.1);
end
```

(movie.avi)

To see the output click next to the shaded block of code. A key step is the axis specification using the **axis** command. This keeps each plot on the same scale as opposed to the automatic scaling MATLAB will do by default.

To generate an avi movie file like the one imbedded into this pdf, the code is more complex and not necessary for CAAM 336.

```
writerObj = VideoWriter('movie.avi');
open(writerObj);
figure;
plot(x,U(:,i));
axis([0 10 0 1.1]);
xlabel('$x$','interpreter','latex','fontsize',16);
ylabel('$u(x,t)$','interpreter','latex','fontsize',16);
title(['t=' num2str(t(i))],'interpreter','latex','fontsize',16);
set(gca,'nextplot','replacechildren');
for i=2:length(t)
    plot(x,U(:,i));
    axis([0 10 0 1.1]);
    title(['t=' num2str(t(i))],'interpreter','latex','fontsize',16);
    frame=getframe(gcf);
    writeVideo(writerObj,frame);
end
close(writerObj);
```

We conclude this section with an example of how to display your data simultaneously with a plot and a color bar. Let us consider the same function $u(x,t)$ plotted over x at sequential times t. Now imagine that $u(x,t)$ represented temperature at position x in a bar at time t. We can display this using the code below. In particular, consider the use of the following commands:

- **axes** allows us to create two axes in a single figure of specified position and size.
- **pcolor** is similar to imagesc and to surf with the view set to directly above. It plots colors corresponding to the size of the input matrix values.
- **kron** is the kronecker tensor product of the two arguments. In the example below it is simply stacking the vectors on top of each other, i.e. kron(x,[1;1])=[x;x]
- **caxis** takes in a vector of length 2 and manually sets the minimum and maximum values for the color scaling
- **set** here takes in gca, the handle to the current axis (think gca=get current axis), and is used here to remove the ytick marks (hence the empty bracket [])

```
figure;
ax1 = axes('position', [.1 .35 .85 .55]);
ax2 = axes('position', [.1 .05 .85 .2]);

for j=1:length(t)
    axes(ax1);
    plot(x,U(:,j),'b-','linewidth',2);
    axis([0 ell 0 1.3*max(U(:,1))]);
    xlabel('$x$','interpreter','latex',...
    'fontsize',16);
    ylabel('$u(x,t)$','interpreter','latex',...
    'fontsize',16);
    title(['$t=$' num2str(t(j))],'interpreter','latex',...
    'fontsize',16);
    axes(ax2);
    pcolor(kron(x,[1;1]),[0 .1*ell],kron(U(:,j)',[1;1]));
    axis equal;
    caxis([0 max(U(:,1))]);
    axis([-0.0 ell -0.02*ell .12*ell]);
    set(gca,'ytick',[]);
end
```

(movie2.avi)

5 Numerical and Symbolic Integration

Another of MATLAB's many tools is the ability to compute integrals of functions both numerically and symbolically. We go highlight this feature through several examples in the sections below.

5.1 Numerical Quadrature: quad

The main MATLAB function for numerical integration is **quad**. The name comes from quadrature which is a historical mathematical term meaning the calculation of area. **quad** takes in a scalar valued function and two limits of integration and computes the integral of the function. The input function can be defined in several ways. First, let's compute

$$\int_0^1 f(x)dx, \qquad \text{where} \qquad f(x) = \cos(\pi x)$$

using an inline function of one variable.

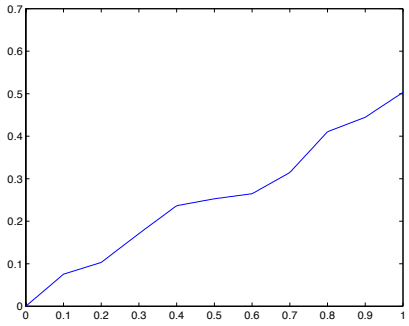
```
>> f=@(x) cos(pi*x);
>> quad(f,0,1); % should equal 0
ans =
   2.7756e-17
```

At times, it may be helpful to integrate a function of two variables for a fixed value of one of the variables. We can do this as follows:

```
>> f=@(x,n) cos(n*pi*x);
>> quad(@(x) f(x,1),0,1) % should equal 0
ans =
   2.7756e-17
>> g=@(x,n) sin(n*x);
>> quad(@(x) g(x,1),0,pi) % should equal 2
ans =
    2.0000
```

We can also apply **quad** to non-inline functions that we've defined in MATLAB. To compute the integral of the piece-wise linear function defined in a previous section, we simply write

```
>> n=10;
>> a=rand(n,1);
>> x=linspace(0,1,200);
>> figure; plot(x,pwlinearfxn(a,x));
>> numInt=quad(@(x) pwlinearfxn(a,x),0,1)
numInt =
    0.2525
```



To double check the value of the integral, we can use simple geometry to compute the area by adding up areas of trapezoids:

$$\int_0^1 h(x)dx = \frac{1}{2n^2}\sum_{j=1}^{n}(2(n-j)-1)a_j$$

where $h(x)$ is our piece-wise linear function and a_j are the slopes over the regions $[(j-1)/n, j/n]$ for $j = 1, \ldots, n$. Computing this in MATLAB, we confirm our result:

```
>> v=2*(n-(1:n))+1;
>> exactInt=1/(2*n^2)*v*a
exactInt =
    0.2525
```

We can even compute integrals of products of functions: small

```
>> quad(@(x) f(x,1).*g(x,1),0,1)
ans =
   -0.1737
```

Note here that we used **.*** instead of simply ***** to multiply the functions. This is necessary because **quad** will use compute $f(x, 1)$ for vector valued x and similarly for g. The computation will not make sense unless we specify that it is element-wise multiplication as opposed to matrix multiplication.

5.2 Symbolic Mathematics: syms

MATLAB also allows you to compute the value of many integrals exactly through their symbolic toolbox. Let's repeat the calculations we did above but this time symbolically. The key functions in MATLAB that we need are **syms**, **int**, and **simplify**.

First we must declare the symbolic variables using **syms**:

```
>> syms x;
```

Next let's define $f(x) = \cos(\pi x)$:

```
>> f=cos(pi*x)
f =
cos(pi*x)
```

To integrate this function, we use the **int** command. First we can compute the indefinite integral.

```
>> int(f)
ans =
sin(pi*x)/pi
```

We can compute a definite integral as well.

```
>> int(f,0,1)
ans =
0
```

Now let's define $f_n(x) = \cos(n\pi x)$ and compute $\int_0^1 f_n(x)dx$.

```
>> syms n;
>> f=cos(n*pi*x)
f =
```

```
cos(pi*n*x)
>> int(f,x,0,1)
ans =
sin(pi*n)/(pi*n)
```

Unfortunately in MATLAB we can't specify n as an integer so MATLAB doesn't know that in fact this is 0 for all $n \in \mathbb{Z}$. Here are other examples.

```
>> g=sin(n*x)
g =
sin(n*x)
>> int(g,0,pi)
ans =
(2*sin((pi*n)/2)^2)/n
>> int(f*g,0,1)
ans =
-(cos(pi*n)*cos(n) + pi*sin(pi*n)*sin(n) - 1)/(n - pi^2*n)
```

Finally at times MATLAB may compute something that it knows how to simplify further. Using the **simplify** command allows you to do this.

```
>> h=sin(n*pi*x)
h =
sin(pi*n*x)
>>f^2+h^2
ans =
sin(pi*n*x)^2 + cos(pi*n*x)^2
>>simplify(f^2+h^2)
ans =
1
```