In this document we’ll go over the basics of MATLAB. It is recommended that you work through this
document with Matlab open, and test each idea in turn.
As you work your way through, don’t be afraid to explore and experiment.

**Entering vectors and matrices**

At its heart, MATLAB is a program for manipulating matrices. An \( n \) by \( m \) matrix is a box of numbers with
\( n \) rows and \( m \) columns. Vectors are a special case where there is only one row (a row vector) or only one
column (a column vector). Try typing

\[
A = [1 \ 2 \ 3]
\]

in the command window and then press the enter key. MATLAB answers with

\[
A = \\
1 \ 2 \ 3
\]

What has happened here? MATLAB has assigned the 1 by 3 matrix (row vector) \([1 \ 2 \ 3]\) to the variable
\( A \). Actually what you saw on the screen probably didn’t look exactly like this, since by default, MATLAB
double spaces all the output. To change this to single spacing type

```
format compact
```

in the command window and then hit enter. The workspace window on the top left shows all the variables
that have been defined. There should be an entry for the matrix \( A \) giving information about the size (1 by
3) and how much computer memory it uses.

When inputing a matrix \( A \) to MATLAB, we define it row by row and use a semi-colon \( ; \) to separate the
rows. For example, the matrix

\[
\begin{pmatrix}
1 & 2 & 3 \\
4 & 5 & 6
\end{pmatrix}
\]

is defined in MATLAB with the command

\[
A = [1 \ 2 \ 3; 4 \ 5 \ 6]
\]

Notice that the semi-colon is placed after the last entry of a row and so first entry of a new row should come
after a semi-colon. Also note that the variable \( A \) has been re-assigned from its previous value. You can also
see that the size of the variable \( A \) has changed in the workspace window.

The semi-colon operator \( ; \) has another useful purpose - it suppresses the output of MATLAB when placed
at the end of the command. For example, typing

\[
A = [0 \ 0.25 \ 0.5 \ 0.75 \ 1];
\]

defines the vector \((0 \ 0.25 \ 0.5 \ 0.75 \ 1)\) in MATLAB, but silently. Try it yourself. This command is quite
useful if you don’t care about the output, especially if it is very long. To check that \( A \) has been defined
correctly, type
MATLAB answers

A =

| 0 | 0.2500 | 0.5000 | 0.7500 | 1.0000 |

confirming that the value of A is again (0 0.25 0.5 0.75 1). Note that MATLAB is case sensitive. Thus, if you type a instead of A, MATLAB will try to access a different variable from A.

The command

clear

clears all the variable assignments. Try it, and note the change in the workspace window. It’s a good idea to type this before starting a new calculation, to make sure that there are no unexpectedly defined variables.

There is a shorthand notation for entering row vectors with evenly spaced entries. Try

\[ X = 1:10 \]

Get the idea? The colon operator : is an important MATLAB operator useful to generate row vectors. It occurs in different forms. Try

\[ X = 0:2:20 \]

The number in the middle — in this case 2 — is the increment. It can be any size, even negative.

If A and B are two matrices with the same number of rows, then \([A B]\) is the matrix obtained by placing them side by side. The same thing works for any number of matrices. In fact the notation \([1 2 3]\) is a special case of this. MATLAB regards numbers as 1 by 1 matrices, and the matrix \([1 2 3]\) is obtained by placing three of these 1 by 1 matrices side by side.

Component-wise operations on matrices

In the course, we have introduced addition and scalar multiplication of matrices. Recall that these are component-wise operations. This means that if we add two matrices X and Y we simply add each component of X to the corresponding component of Y. Notice that in order to add two matrices, their dimensions must be identical. Define X to be \([1 2 3; 4 5 6]\) and Y to be \([1 1 1; 1 1 1]\). Then add X and Y by typing

\[ X+Y \]

Subtraction works exactly the way you would expect it to, for example the command

\[ X-Y \]

subtracts Y from X. Notice that if the result of a MATLAB calculation is not explicitly assigned to a variable, it gets assigned to the variable ans by default.

Scalar multiplication is also a component-wise operation. According to the definition, to multiply the scalar s times a vector (or matrix) X we must multiply each component of X by s. Try

\[ 2*X \]

Check that \(X+X\) gives the same answer.

In MATLAB, if you have a vector or matrix A, then MATLAB understands that adding a scalar to A means that you would like to add that scalar to every entry of A. So, for instance, \([1 3] + 2\) returns the matrix (vector) \([3 5]\).
Changing part of a matrix

Suppose \( A \) is a MATLAB matrix. We say a matrix is \( n \times m \) if it has \( n \) rows and \( m \) columns. Consider the following \( 3 \times 4 \) matrix \( A \) as an example:

\[
A = \begin{bmatrix}
1 & 2 & 1 & 2 \\
3 & 4 & 3 & 4 \\
5 & 6 & 5 & 6
\end{bmatrix}
\]

How can we extract pieces of \( A \)? Individual entries can be accessed by specifying the row \( n \), counting down from the top, and column \( m \), counting from the left, in the notation \( A(n,m) \). For example:

\[
A(2,3) = 7
\]

We can change the entry to, say, 7 by entering the command:

\[
A(2,3) = 7
\]

Notice than when we adjust a component of \( A \) and omit the semi-colon at the end of the command, MATLAB displays the new version of \( A \).

To extract a sub-matrix, we must specify a range of rows and columns. For example, (assuming you have changed the \( A(2,3) \) entry to 7):

\[
A(1:2,2:3)
\]

Now let’s set all these entries to zero.

\[
A(1:2,2:3) = [0 \ 0; \ 0 \ 0]
\]

By the way, we could have used \texttt{zeros(2,2)} in place of \[0 \ 0; \ 0 \ 0\], i.e. used the command:

\[
A(1:2,2:3) = \text{zeros}(2,2)
\]

The function \texttt{zeros(m,n)} is built into MATLAB which creates a \( m \times n \) matrix with all zero entries. A similar function is \texttt{ones(m,n)}, except that \texttt{ones} creates a \( m \times n \) matrix with all unit entries. So if we wanted to turn the entries of \( A \) previously changed to zeros into ones, we could use the command:

\[
A(1:2,2:3) = \text{ones}(2,2)
\]
A =
1 1 1 2
3 1 1 4
5 6 5 6

Remember you can get information about any function in MATLAB, like ones, by typing help ones.

When specifying ranges of rows and columns, we may use end to denote the last row or column. This is handy if we are not sure how long the matrix is. Also, we can mix the notations for single rows or columns, and ranges. For example, to extract the first row of A, that is, A(1:1,1:4) we could equally well use A(1,1:end).

Even more succinctly, the range 1:end can simply be specified by :.

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

This a handy notation if you want to perform operations on a single row or column of a matrix. You may want to modify this matrix so that the second row is the result of subtracting the first row from the second. This is achieved by

```
>> A(2,:) = A(2,:) - A(1,:)
A =
1 1 1 2
2 0 0 2
5 6 5 6
```

Try it. You may have already learnt in the lectures, or you will soon, that performing operations on the individual rows of a matrix is a common technique to find the solution of a linear system.

**Matrix-vector and matrix-matrix multiplication**

The single asterisk * denotes matrix-vector or matrix-matrix multiplication in MATLAB. The matrices must have the appropriate dimensions for this to take place, otherwise you will get an error message. As an example, try defining

```
M = [
  0 1
  3 4
] 
N = [
  1 0
  2 2
].
```

Now compute M*N and N*M. Do you get the same answer? Should you get the same answer?

To multiply an appropriately-sized MATLAB column vector x by a matrix M, you similarly write M*x. Check you can do this with MATLAB: if \( \bar{x} = \begin{pmatrix} -1 \\ 1 \end{pmatrix} \), compute MN\( \bar{x} \). You should get \( \bar{x} = \begin{pmatrix} 0 \\ -3 \end{pmatrix} \). Can you verify that by hand as well?

*Dangerous Note*: If you really want to do a component-wise multiplication of two matrices of the same size, MATLAB will comply, provided we use the operator .\* (a dot followed by a single asterisk). Try

```
X.*X
```

and check that each component of the result is the product of the corresponding components of X. Why is this dangerous? Because it is important that no-one confuses this component-wise multiplication with proper matrix multiplication! You are unlikely to need to do component-wise multiplication in this course.
Functions

MATLAB has a large number of built in functions. Let us explore just a few. Let’s begin with the dot product. If X and Y are vectors, then \texttt{dot(X,Y)} computes the dot product of X and Y. Try some simple examples. Does it matter whether X or Y are row or column vectors? Recall that a function can be thought of as a device that takes some input and uses it to compute some output. For the \texttt{dot(·,·)} function, the input is a pair of vectors and the output is a number.

If you want to more information about a MATLAB function, you can use the built in help. If you know the name of a function, you can type

\texttt{help <functionname>}

at the command prompt to get information about that function. For example, to find information about the function \texttt{dot}, type

\texttt{help dot}

in the command window and information about how to use the function will appear.

For example, the common trigonometric functions \texttt{sin}, \texttt{cos} and \texttt{tan} are available in MATLAB. The inverse functions are called \texttt{asin}, \texttt{acos} and \texttt{atan}. Normally, these function take a real number as input and produce another real number as output. However, in MATLAB, if they are applied to a vector or a matrix, MATLAB computes the functions component-wise. For example, if \( X = \begin{bmatrix} 1 & 2 & 3 \end{bmatrix} \) then \( \sin(X) = \begin{bmatrix} \sin(1) \\ \sin(2) \\ \sin(3) \end{bmatrix} = \begin{bmatrix} 0.8415 \\ 0.9093 \\ 0.1411 \end{bmatrix} \).

\texttt{det}

The function \texttt{det(A)} returns the determinant of a square (same number of rows and of columns) matrix A. You have seen in the lectures how to compute determinants of \( 2 \times 2 \) (2 rows and 2 columns) matrices. Later in the course you will see how to calculate determinants of larger, square matrices by hand but it can be done now using MATLAB.

Solving linear systems of equations in Matlab

Row Reduction

By applying the process of Gaussian Elimination (row reduction) to a system of linear equations, it is reduced to a form where its solutions are easily characterized. This form is called Reduced Row Echelon Form. While the row operations are easy to perform by hand on small systems, as more equations are introduced into the system it makes more sense to automate the procedure with a computer. In MATLAB, the command \texttt{rref(A)} outputs the Row Reduced Echelon Form of a matrix A.

For example, to find the solutions to

\[
\begin{align*}
2x_1 & +3x_2 & -x_3 & +4x_4 & = 0 \\
-x_1 & - x_2 & + x_3 & + x_4 & = 2 \\
2x_1 & - 2x_2 & + x_3 & + x_4 & = 1
\end{align*}
\]

in MATLAB, we first input the augmented matrix for this problem,

\[
A = \begin{bmatrix} 2 & 3 & -1 & 4 & 0 \\ 1 & -1 & 1 & 1 & 2 \\ 2 & -2 & 1 & 1 & 1 \end{bmatrix}
\]
Then
\[
\text{rref}(A)
\]
produces the output
\[
\text{ans} = \\
\begin{bmatrix}
1 & 0 & 0 & 1 & 0 \\
0 & 1 & 0 & 1 & 1 \\
0 & 0 & 1 & 1 & 3 \\
\end{bmatrix}
\]
from which we can read off
\[
\begin{align*}
x_1 + x_4 &= 0 \\
x_2 + x_4 &= 1 \\
x_3 + x_4 &= 3 \\
\end{align*}
\]
We now find all solutions to the original system following the procedure learnt in the lectures. We take \( x_4 = s \) to be a parameter and find the general solution
\[
\begin{bmatrix}
x_1 \\
x_2 \\
x_3 \\
x_4 \\
\end{bmatrix} = \\
\begin{bmatrix} 0 \\ 1 \\ 3 \\ 0 \\
\end{bmatrix} + s \begin{bmatrix} -1 \\ -1 \\ -1 \\ 1 \end{bmatrix}
\]

Using the matrix inverse

The Matlab command to find the inverse of a matrix \( A \) is \text{inv}(A). Obviously, we can use this to solve \( n \times n \) systems in the form \( A\vec{x} = \vec{b} \) by setting \( \vec{x} = \text{inv}(A) \vec{b} \). However, this method is computationally inefficient for large matrices and should be avoided if possible. Also, it will not work if the matrix \( A \) is not invertible (the case when there are zero or infinitely many solutions).

Using backslash

The most efficient way to solve systems with a unique solution is to use the backslash operator. For instance, to solve \( A\vec{x} = \vec{b} \) you would type \( \vec{x} = A\backslash \vec{b} \). The backslash command is the method of choice when there is a unique solution – otherwise, use \text{rref} as described above.

Creating a .m MATLAB file and plotting

In this section, we will learn how to write a script and plot functions.

Plotting

The \( \text{x} = \text{linspace}(a,b,N) \) command in MATLAB creates a \( 1 \times N \) vector \( x \) whose entries begin at \( a \) in the first element and increase in equal increments up to \( b \) in the \( N \)-th element. Try typing \( x = \text{linspace}(0,1,4) \) into the command line. The answer should look like:
\[
0 \\ 0.3333 \\ 0.6667 \\ 1.0000
\]
To plot the function \( y = x \), first create a vector named \( y_1 \) by typing \( y_1 = x; \). Next, type \( \text{plot}(x,y_1); \). A figure should appear, and the function \( y = x \) should be displayed. Next, create a vector named \( y_2 \) and assign it the value of \( x^2 \) by typing \( y_2 = x.^2 \). To avoid clearing the previous plot, type \text{hold on}; then type
plot(x,y2,'r'); If the command hold on is not entered, the previous plot will be erased and replaced with the new one. The additional argument ‘r’ colors the line red. The figure should now display two plots - one of \( y = x \) and \( y = x^2 \). Notice that the \( y = x^2 \) plot is rather jagged - this is because we are lacking spatial resolution. Remember that the vector \( x \) only has 4 elements. Try repeating this procedure with more elements in \( x \). The more elements it has, the higher the spatial resolution, and the better the plot will look.

Label the \( x \) and \( y \) axes by typing xlabel(‘x’,’FontSize’,14); ylabel(‘y’,’FontSize’,14). Set the font size of the axis labels by typing set(gca,’FontSize’,14). If you want to see a grid, type grid on. A legend can be created by typing the command legend(‘y = x’, ‘y = x^ 2’).

Creating a .m file

When there is a long list of commands that we must enter in order to accomplish a certain task, typing them into the command line and executing them one by one is slow and clumsy. In a script, we enter all the commands at once, each on a separate line, which MATLAB then executes in order. To open a script window, click “File” at the top left corner, then “New,” then “M-file.” A new window should open - it is essentially a text editor. In this editor, you can enter all the commands in the previous subsection on separate lines (be sure to end each line with a semicolon to suppress output). It is usually good practice to begin a script with the commands “clear all” and “close all” (on separate lines). The former clears all variables from the memory, while the latter closes all plot windows. Your script should begin like this:

```matlab
clear all
close all
x = linspace(0,1,100);
y1 = x;
y2 = x.^ 2;
...etc
```

Nothing is needed to indicate the end of the script. When you are done, press “F5” to run the script. The result should be a plot of \( y = x \) in blue and \( y = x^2 \) in red.

For loops

For loops are used when a sequence of tasks needs to be performed many times. A for loop in Matlab is written as the following:

```matlab
for k = 1:1:10
    {tasks}
end
```

The first line says that the loop is indexed by the variable \( k \), which takes on integers from 1 to 10, incremented by 1 (note the similarity of this notation to that introduced in the previous lab for creating vectors using colons). This for loop will run 10 times, each time for a larger value of \( k \). The \{tasks\} are commands to be executed by MATLAB each time through the loop. The end tells MATLAB that anything after the word end is not part of the for loop. To illustrate this, let us write a script to sum the numbers 1 through 20 while displaying the value of the loop index each time through the loop:

```matlab
clear all
close all
N = 20;
s = 0;
for k = 1:1:N
disp(['the value of k is ', num2str(k)]);
s = s + k;
end
disp(['the sum is ', num2str(s)]);
```
disp(['the value of k after the loop completes is ', num2str(k)]);

The disp() function displays strings to the command window. The num2str() function converts a number to a string so that it can be displayed in disp(). The square brackets concatenate the two strings. Variables can be reassigned inside a for loop. The following script outputs the first 100 numbers in the Fibonacci sequence:

clear all
close all
N = 98;
s = 1;
s_prev = 0;
disp(num2str(s_prev));
disp(num2str(s));
for k = 1:1:N
    s_next = s + s_prev;
disp(num2str(s_next));
s_prev = s;
s = s_next;
end

Notice the order in which the variables were reassigned inside the for loop so as to not overwrite values that are still needed. WARNING: MATLAB does not protect the names of its built-in functions. For example, sum() is a MATLAB function that sums components of a vector or matrix, but you are allowed to use it as a variable name. As soon as you assign a value to sum, the function is no longer accessible. This can be the cause of bugs that can be extremely difficult to catch. As such, do not knowingly use variable names that are names of MATLAB functions.

Random numbers, if statements, and while loops

In this section we will learn how to use MATLAB's random number generator, if statements, and while loops.

The rand function

The rand function in MATLAB generates a (pseudo)random number uniformly distributed between 0 and 1. Try typing rand into the command line (without the semicolon) a few times. Notice that it outputs a different number each time. For the purposes of this lab, we only need to know that the probability that rand produces a number that is less than 1/2 is 1/2. That is, if you ask rand to generate a sequence of 10000 numbers (rand(1, 10000)), approximately half the numbers will be less than 1/2, and approximately half will be greater than 1/2.

If else statement

MATLAB's if else statement is written:

if {Boolean expression}
    {perform task 1}
else
    {perform task 2}
end

A Boolean expression evaluates to either true or false. If the Boolean expression is true, task 1 is executed.
Otherwise, task 2 is executed. The most common Boolean operators are less than (<), greater than (>), equals (==), and not equal to (~=). Try a few on your own. Type \(2 < 3\), \(2 > 3\), \(2 == 3\), and \(2 \sim= 3\) into the command line and note the answers. Following convention, 0 is false, and 1 is true.

Two logical conjunctions are also very useful: logical and \(&&\) and logical or \(||\). They take two Boolean expressions and output a true or false value according to the regular rules of logical and and logical or. Try some one your own: \((2 < 3) || (1 > 0)\), \((2 < 3) && (1 > 0)\), etc.

Now let us use the if else statement to see if the \texttt{rand} function indeed does return a number less than 1/2 with probability 1/2:

```matlab
clear all
close all

more = 0;
less = 0;
N = 10000;

for k = 1:1:N
    if rand < 1/2
        less = less + 1;
    else
        more = more +1;
    end
end

disp(['less than 1/2: ', num2str(less)]);
disp(['greater than 1/2: ', num2str(more)]);
disp(['ratio: ', num2str(less/more)]);
```

Notice that the first \texttt{end} ends the if else statement, and the second \texttt{end} ends the for loop. Notice also that we take \(N\) very large so that the ratio will always be very close to unity. Try it a few times with \(N = 10\) - you will notice that sometimes the ratio deviates quite a bit from unity.

**While loops**

While loops are similar to for loops, except while loops are used when it is unknown how many iterations are needed. In MATLAB, they are written

```matlab
while \{Boolean expression\}
\{perform tasks\}
end
```

A while continues to repeat and perform the tasks specified until the Boolean expression evaluates to false. As soon as the Boolean expression evaluates to false, the while loop is terminated. If the Boolean expression never evaluates to false, the while loop iterates forever (in which case you need to hit \texttt{ctrl} \texttt{c} to break out of the loop).

**Creating large matrices using MATLAB functions**

We already know that \texttt{eye(n)} creates the \(n \times n\) identity matrix. What if we wanted a matrix whose diagonal above or below the main diagonal is all ones? We do not want to enter the entries one by one. Here, we will
use the diag() and ones() functions. Try typing

```
N = 9;
diag(ones(1, N), 1)
```

Notice that this creates a $10 \times 10$ matrix (one larger than $N$) whose entries on the diagonal above the main is all 1. If you want a matrix whose diagonal entries below the main is all $-1/2$, type

```
N = 9;
diag(-0.5*ones(1, N), -1)
```

To created a “banded” matrix, we simply need to add matrices together (ensuring they are of the same dimension of course):

```
N = 10;
M1 = diag(ones(1, N-1), -1);
M2 = diag(ones(1, N-1), 1);
M = -2*eye(N) + M1 + M2
```

### Eigenvalues and Eigenvectors

If $A$ is an $n \times n$ matrix, then $\lambda$ is called an eigenvalue of $A$ with eigenvector $v$ if $Av = \lambda v$. A typical matrix $A$ will have $n$ eigenvalues $\lambda_1, \lambda_2, \ldots, \lambda_n$ and $n$ corresponding eigenvectors $v_1, v_2, \ldots, v_n$. Some of the eigenvalues may be repeated, i.e., $\lambda_i = \lambda_j$ even though their eigenvectors $v_i$ and $v_j$ are different. There are also exceptional matrices which have fewer than $n$ linearly independent eigenvectors.

Eigenvalues and eigenvectors are of central importance in linear algebra and its applications, so it is not surprising that MATLAB is very good at finding them. Let’s start with the basic commands for finding eigenvalues and eigenvectors. Let $A$ be an $n \times n$ matrix. If we only care about the eigenvalues then we can use

```
eig(A)
```

This returns a vector of length $n$ containing the eigenvalues. For example, to find the eigenvalues of the matrix

$$A = \begin{pmatrix} 1 & 2 & -1 \\ 1 & -1 & 1 \\ 3 & -2 & 1 \end{pmatrix}$$

we enter the following:

```
>> A = [ 1 2 -1; 1 -1 1; 3 -2 1];    
>> eig(A)                           
ans =                                   
 1.4780  
-0.2390 + 1.6277i  
-0.2390 - 1.6277i
```

and read off the three eigenvalues

$$\lambda_1 = 1.4780, \quad \lambda_2 = -0.2390 + 1.6277i, \quad \lambda_3 = -0.2390 - 1.6277i.$$ 

where $i = \sqrt{-1}$. Note that two of the three eigenvalues are complex. If we want both the eigenvalues and eigenvectors, then

```
[V,D] = eig(A)
```
puts the eigenvectors in the columns of the $n \times n$ matrix $V$ and the corresponding eigenvalues on the diagonal of the $n \times n$ matrix $D$. The off diagonal entries of $D$ are set to zero. Note that MATLAB returns eigenvalues that are normalized (have unit length).

Let’s try this with a $3 \times 3$ matrix. Define $A$ as

\[
>> A = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 3 & 4 \\ 3 & 4 & 5 \end{bmatrix}
\]

\[
A =
\begin{bmatrix}
1 & 2 & 3 \\
2 & 3 & 4 \\
3 & 4 & 6
\end{bmatrix}
\]

Notice that this is a symmetric matrix as $A^T = A$. A well known property of symmetric matrices is that their eigenvalues are real and their eigenvectors are orthogonal. Let’s calculate the eigenvalues and eigenvectors of $A$ with MATLAB and confirm the eigenvalues are real and the eigenvectors are orthogonal.

\[
>> [V,D] = \text{eig}(A)
\]

\[
V =
\begin{bmatrix}
-0.9269 & 0.0856 & 0.3653 \\
0.1307 & -0.8389 & 0.5283 \\
0.3517 & 0.5374 & 0.7665
\end{bmatrix}
\]

\[
D =
\begin{bmatrix}
-0.4203 & 0 & 0 \\
0 & 0.2336 & 0 \\
0 & 0 & 10.1867
\end{bmatrix}
\]

Notice that the eigenvalues (entries along main diagonal of $D$) are all real numbers. Let’s confirm the first column of $V$ is really an eigenvector. To do this we take the first column of $V$ and put it in a vector $X$.

\[
X = V(1:3,1);
\]

Next we define \texttt{lambda} to be the corresponding eigenvalue.

\[
\text{lambda} = D(1,1);
\]

Now compare $A*X$ with \texttt{lambda*X};

\[
>> A*X
\]

\[
\text{ans} =
\begin{bmatrix}
0.3896 \\
-0.0550 \\
-0.1478
\end{bmatrix}
\]

\[
>> \text{lambda*X}
\]

\[
\text{ans} =
\begin{bmatrix}
0.3896 \\
-0.0550 \\
-0.1478
\end{bmatrix}
\]

Reassuringly, they are identical. We can also check that the first two eigenvectors are orthogonal. Set $Y = V(1:3,2)$; Now try

\[
dot(X,Y)
\]

The answer should be zero\(^1\).

\(^1\)Note that MATLAB approximates floating point numbers, so an exact answer of zero may be computed with size $10^{-14}$. 

11