Lab 1 - Introduction to Matlab

Contents

1 Linear Algebra in MATLAB

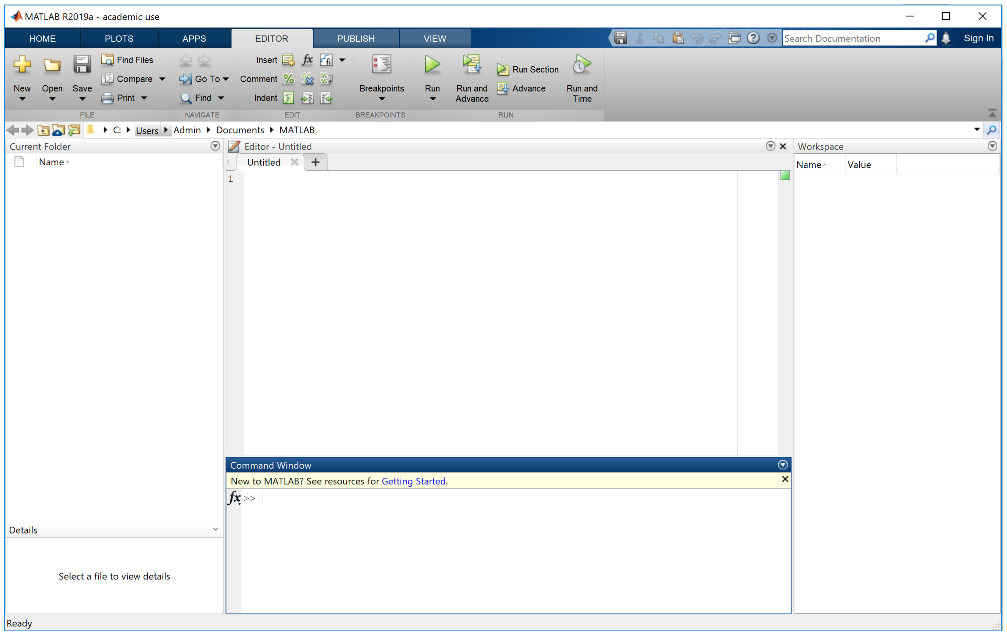
MATLAB, or the MATrix LABoratory, has been built from the ground up to run matrix computations quickly and efficiently. Although most computationally intensive tasks are eventually ported to C, MATLAB serves as a rapid prototyping language; easier than C and able to provide a quick proof of concept. In addition, its many plugins mean that MATLAB is an out of the box solution for many scientific computing and engineering tasks, from solving differential equations to signal processing, to interfacing with ARDUINO boards.

In this first tutorial, we will run through the basics of linear algebra in MATLAB. We will talk about visualizing data, constructing matrices, and using matrices as linear transforms, and using matrices to solve equations. This lab consists of two parts: A follow along coding section where with exercises that will not be turned in, and a set of questions. **If you are already familiar with MATLAB, you may skip to section 2 where we begin solving linear systems.**

To install an activate MATLAB as a Northeastern student, follow the information found here: `https://northeastern.service-now.com/kb_view.do?sysparm_article=KB0012568`. You only need to download **MATLAB** and **SIMULINK** for this course, but you may add any other packages your are interested in.

1.1 Introduction:

Upon opening MATLAB, you'll be greeted by a four part screen as below:



New to MATLAB? See resources for Getting Started.

```
>>
```

Select a file to view details

This screen contains

- **Current Folder**: The directory where your scripts and data will be stored.
- **Editor**: For editing scripts and storing code. If this does not appear click "New" and then "Script."
- **Workspace**: A list of the currently initialized variables.
- **Command Window**: A place to enter single commands, like one a calculator.

First steps: MATLAB as a giant calculator At its simplest, MATLAB's command window can be used as a giant calculator. To start, enter

```
>> 1+1
```

in the command window and press **Enter**. Two things will happen: First, MATLAB will return

```
ans =
     2
```

and second a new variable will appear in the workspace called `ans` with the value 2. The `ans` variable will always contain the result of the last computation performed on the command window.

If you want to store the result of a computation for later use, you can assign a variable to the result using = as so:

```
>> myvar = 5 + 7 + 9
```

After you hit Enter, `myvar` will appear in the workspace containing the result of the calculation `5 + 7 + 9`. If you may then use `myvar` in any future computation

```
>> myvar*8
```

or assign a new value to it `MATLAB >> myvar = 5 + 7 + 9` ``````` `MATLAB >> myvar = 400/20` or both

```
>> myvar = myvar/100
```

Question: What are the results of each of the computations above? If you perform them in a different order, do you get the same result?

Operators and Order of Operations: MATLAB has the following common operators:

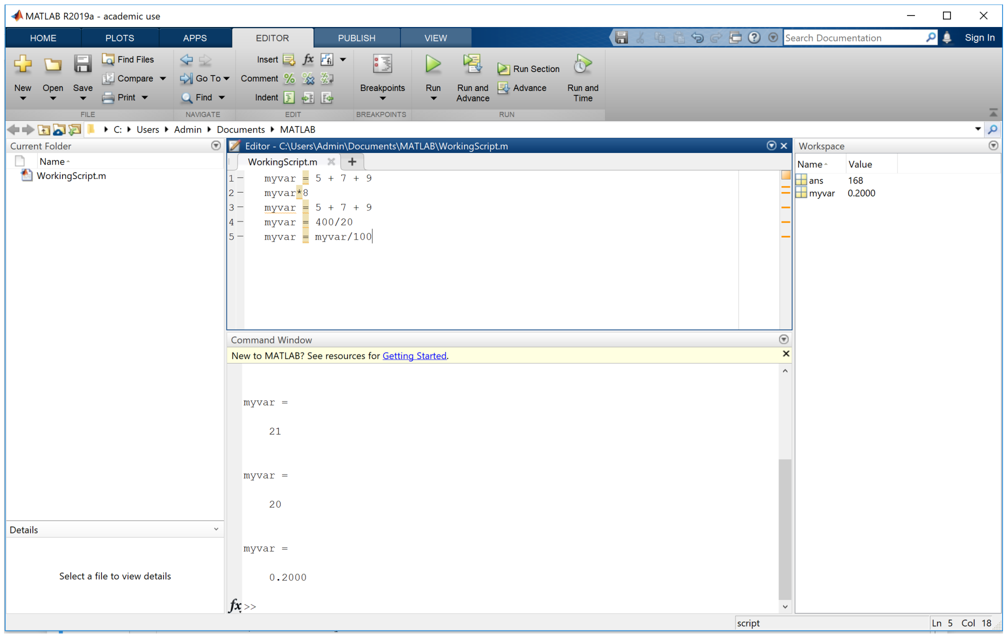
Symbol	Name
+	Addition
-	Negation and Subtraction
*	Multiplication
/,\	Division
-	Exponentiation

The precedence of the order of operations from highest to lowest is `()`, `^`, `-` negation, `*`,`/`,`\`,, then `+`,`-`. Take a moment to think about how

```
>> -10^-4/2+(10+2^1*2)
```

will evaluate, then check your answer.

Command Window vs Editor While all commands can be typed into the command window, it's often better to use the script editor for longer scripts. Click **New** and then **Script** to bring up a blank script. All commands can be typed into the script editor, and then run all together by pressing **Run**.



```
myvar = 5 + 7 + 9
myvar*8
myvar = 5 + 7 + 9
myvar = 400/20
myvar = myvar/100
```

```
myvar =

    21


myvar =

    20


myvar =

    0.2000

>>
```

Name	Value
ans	168
myvar	0.2000

1.2 Vectors

A **vector** in MATLAB is just a list of numbers separated by spaces or `,`'s for **row vectors**,

```
>> vec1 = [1 2 3 4]
>> vec2 = [7, 10, 30, 50]
```

or by semicolons `;` for **column vectors**,

```
>> vec3 = [-1; -4; -9; -16]
```

A column vector can be switched to a row vector (and visa versa) by **transposition** using the apostrophe `'`

```
>> vec3'

ans =

    -1    -4    -9   -16
```

You can access the elements of a vector using their position starting at 1, so `vec2(1)` is `7` and `vec3(4)` is `16`.

Vector Operations Vectors of compatible dimensions can be added

```
>> vec1 + vec2
```

multiplied by scalars

```
>> vec1*4
```

have a scalar uniformly added to them

```
>> vec2 + 4
```

Note that trying to add vectors of incompatible dimensions produces something strange:

```
>> vec1 + vec3

ans =

     0     1     2     3
    -3    -2    -1     0
    -8    -7    -6    -5
   -15   -14   -13   -12
```

What has happened here is that MATLAB is constructing a matrix by adding each row entry of `vec1` to the each column of `vec3`. This is rarely used, but can be a convenient way to construct a matrix.

What about vector multiplication? MATLAB is built for matrices, so it will always assume that multiplication like `vec1*vec2` is **matrix multiplication**. Try running

```
>> vec1 * vec2
```

and note the error that you get: `Error using * Incorrect dimensions for matrix multiplication.` It's telling you exactly what has gone wrong, that is that your matrix multiplication has incorrect dimensions. The error messages in MATLAB are quite informative, always remember to read them.

Finally, if we want component-wise multiplication we need to use a `.*`:

```
>> vec1 *. vec3
```

Question: With `vec1`, `vec2` and `vec3` defined as above, which multiplications, for example

```
>> vec1 *. vec2
>> vec1 * vec2
```

will produce valid results? Check you answer with MATLAB.

1.3 Matrices

(See also: https://www.mathworks.com/help/matlab/learn_matlab/matrices-and-arrays.html)

A **matrix** in MATLAB is just a two dimensional vector, and is defined by specifying it's entires row by row. For example, the 2×3 matrix

$$\begin{pmatrix} 1 & 3 & 5 \\ 7 & 9 & 11 \end{pmatrix}$$

can be constructed in MATLAB by

```
>> mat1 = [1,3,5;7,9,11]
```

It is also common to construct a matrix as a vector of vectors:

```
>> mat2 = [[2,4,6];[8,10,12]]
```

As before, we can specify the elements by their position so `mat2(1,1)` is 2, while `mat2(2,3)` is `12`.

Given a matrix, we can get the number of elements using the `numel` function and the dimensions using the `size` function:

```
>> numel(mat1)
>> size(mat1)
```

Note that size itself returns a vector, a 1×2 row vector `[COLUMNS, ROWS].`

MATLAB allows the basic matrix operations on a matrix `M`:

- `M*` - Matrix multiplication, provided dimensions are compatible.
- `M.*` - Component-wise multiplication, provided dimensions are compatible.
- `M'` - Matrix transpose.
- `inv(M)` - Matrix inversion, provided matrix is square and invertible.

For example, given our matrices above we can compute the inverse of `mat1` times the transpose of `mat2` by

```
>> A = inv(mat1 * mat2')
```

Question: What is the result of multiplying `A` by `mat1`?

1.4 Constructing Vectors and Matrices

MATLAB has some builtin ways to make matrix construction easier:

Sequences The code `3:10` will return the vector containing all integers from 3 to 10 as a row vector:

```
>> vec1 = 3:10
```

Similarly, the code `3:4:20` will count starting at 3 and adding 4 each time until it is above 20, returning a vector of all the numbers in the count *less than or equal to* 20. In this case,

```
>> vec2 = 3:4:20
vec2 =

     3     7    11    15    19
```

It does not contain 20, because 20 is not a multiple of 4 more than 3, 20 just serves as an end point. However,

```
>> vec3 = 0:4:20
vec3 =

     0     4     8    12    16    20
```

will contain 20.
We can also count down, for example

```
>> vec4 = 20:-2:1
```

returns the numbers we get when, starting from 20, we add -2 until we are less than 1.

Matrix Construction Functions MATLAB also has some basic matrix construction functions:

- `zeros(N,M)` - Produces an N by M matrix of 0's.
- `ones(N,M)` - Produces an N by M matrix of 1's.
- `rand(N,M)` - Produces an N by M matrix of uniformly distributed random numbers between 0 and 1.

Additionally, you can use sequences to build matrix columns. For example, we can make the matrix

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 4 & 6 & 8 & 10 \\ 4 & 4 & 4 & 4 & 4 \end{pmatrix}$$

by

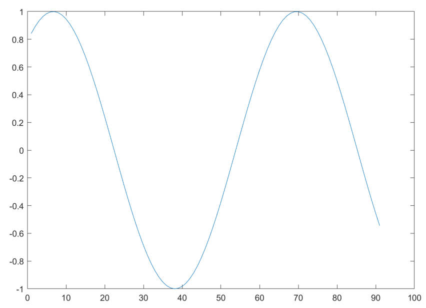
```
>> mat = [1:5 ; 2:2:10 ; 4*ones(1,5)]
```

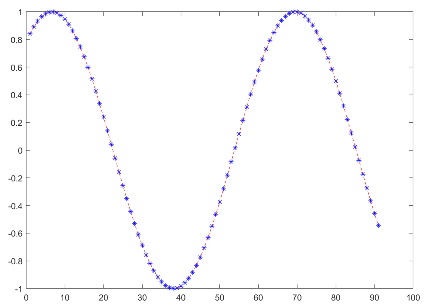
Question: Try using matrix addition, multiplication, and the matrix construction functions above to build * A 2×5 matrix with -1 in the first row and the odd numbers starting at 5 in the second row. * A random matrix with all numbers between 20 and 30. * (Challenge) Multiply two vectors to produce a 100 by 100 matrix

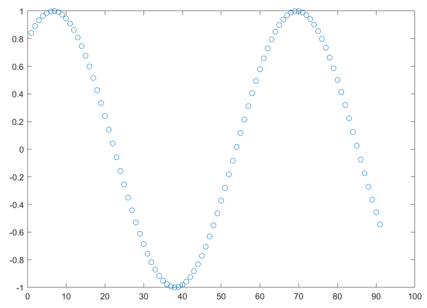
$$\begin{pmatrix} 1 & 2 & 3 & \\ 2 & 4 & 6 & \dots \\ 3 & 6 & 9 & \\ & \vdots & & \ddots \end{pmatrix}$$

1.5 Visualization with MATLAB

MATLAB's `plot` function can be used to display a wide verity of 2D visual information, and we will use it frequently in this course to generate graphs and display linear transforms. The `plot` function displays a sequence of points in 2d connected by lines. The markers for each point and the line styles can be edited, or turned off completely.







To plot points (x_1, y_1), (x_2, y_2), ..., (x_n, y_n), we must supply the plot function with a list of the x-values, `X = [x_1, ..., x_n]` and a list of the y-values `Y = [y_1, ... , y_n]`. For example, to plot a graph of a function like sine or cosine from 1 to 10 we write

```
>> X = 1:10
>> Y = sin(X)
>> plot(X,Y)
```

Question: Looking at `X` and `Y`, we are only plotting 10 data points: $(1, \sin(1))$, $(2, sin(2))$, etc. Make the plot smoother by plotting all the points from 1 to 10 up to two decimal places, for example $(1.01, \sin(1.01))$.

The plot function has many options the are detailed in the documentation (https://www.mathworks.com/help/matlab/ref/plot.html). All attributes can be explicitly defined using property flags like `LineWidth`, but many can also be defined using a character vector, with

Line Style	Description	Line Style	Description
-	Solid line (default)	--	Dashed line
:	Dotted line	-.	Dash-dot line

Marker	Description	Marker	Description
o	Circle	+	Plus sign
.	Point	x	Cross
d	Diamond	^	Upward-pointing triangle
>	Right-pointing triangle	<	Left-pointing triangle
h	Hexagram		

Color	Description	Color	Description
y	yellow	m	magenta
r	red	g	green
w	white	k	black

For example, to plot a red dashed line with crosses for markers we would use

```
>> X = 1:10
>> Y = sin(X)
>> plot(X,Y,'r--x')
```

If you don't specify a line style, MATLAB will not plot the line.

```
>> X = 1:10
>> Y = sin(X)
>> plot(X,Y,'rx')
```

Question: Plot cosine from $-\pi$ to π with blue point markers and no connecting lines. You may use `pi` for the value of π.

Titles and Labels To make our plots readable, we should always include at least a title, and if applicable axis labels and a legend.

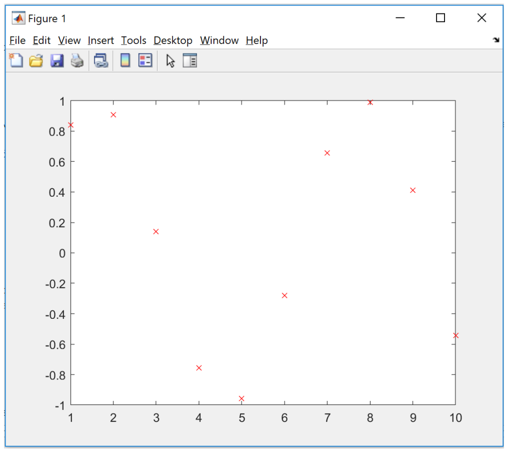
- `title('Plot Title')` - Add a plot title.
- `xlabel('X-Axis Label')` - Add a label to the horizontal axis.
- `ylabel('Y-Axis Label')` - Add a label to the vertical axis.

For example, use

```
>> plot(X,Y,'rx')
>> title('Cosine Function')
>> xlabel('Independent Variable')
>> ylabel('Dependant Variable')
```

to plot and label the cosine function.

Figures and Multiple Plots Each plot in MATLAB lives inside a **figure**. A figure is the window containing the plot.



The `figure` function creates a new figure. You can create multiple figures if you want to generate multiple plots, or you can plot multiple things to the same figure. For example, to plot two functions on different axis, we use

```
>> f1 = figure
>> plot(X, sin(X))
>> title("Sine")
>> f2 = figure
>> plot(X, cos(X))
>> title("Cosine")
```

After we create a new figure, all of the actions we perform will be on that figure. If we want to return to the figure with $\sin(x)$ and change something there, we use `figure(f1)` to reactive it:

```
>> f1 = figure
>> plot(X, sin(X))
>> title("Sine")
>> f2 = figure
>> plot(X, cos(X))
>> title("Cosine")
>> figure(f1)
>> xlabel('The Domain')
```

You may have noticed that each time we call `plot` it erases the previous plot. What if we want to plot two charts on the same axis? There are two options: We can specify multiple charts within the plot function by listing them as `plot(x,f_1(x), x, f_2(x),...)`

```
>> f1 = figure
>> plot(X, sin(X), X, cos(X))
```

or use `hold on` to hold the current contents of the axis while we draw to it:

```
>> f1 = figure
>> hold on
>> plot(X, sin(X))
>> plot(X, X.^2)
>> hold off
```

In the above `hold off` tell MATLAB to stop holding the contents of the current axis. Note that `X.^2` computes x^2, the `.^` tells MATLAB to exponentiate each component instead of trying to exponentiate as a matrix. For more examples take a look here (https://www.mathworks.com/help/matlab/creating_plots/combine-multiple-plots.html)

Question: Plot the function $x/(x^2+1)$ and the function $\cos(1/x)$ from 1 to 4 on the same plot.

1.5.1 2D Plotting

In addition to plotting graphs of functions, we can perform 2D plotting by specifying the x and y coordinates of points. For example, we can draw the diamond with points at (1,0), (0,1), (-1,0), (0,-1) by writing the x coordinates in one vector `X` and the y coordinates in another vector `Y` and passing them to the plot function:

```
>> X = [1,0,-1,0]
>> Y = [0,1,0,-1]
>> plot(X, Y)
```

Notice that MATLAB doesn't complete the square because we didn't tell it to connect the last point back to the first. To do so we just add another copy of $(1,0)$ at the end:

```
>> X = [1,0,-1,0,1]
>> Y = [0,1,0,-1,0]
>> plot(X, Y)
```

Question: Plot a 3x3 grid with no connective lines and circle markers. Can you find a clever way to define the vectors `X` and `Y` that makes the code simple?

Axis Boundaries Often, it's alright to let MATLAB figure out the axis boundaries itself, but (as you may have noticed in the question above) sometime we want to choose the axis boundaries ourselves. To do so, we have use `axis` function:

- `axis( [x_min, x_max, y_min, y_max] )` - Sets the axis so that the horizontal view runs between $[x_min, x_max]$ and the vertical view runs between $[y_min, y_max]$.

For example, to display our square on a scale with x from -2 to 2 and y from -1 to 1, we write

```
>> X = [1,0,-1,0,1]
>> Y = [0,1,0,-1,0]
>> plot(X, Y)
>> axis( [-2,2,-1,1] )
```

Labels With 2D plotting we often want to add labels. The labels support some latex and can be used to annotate points:

- `text(x,y,'My Text','left')` - Add the text `My Text` with the left anchor at the point (x, y). You may specify `'right'` or `'center'` as well for alignments, and if nothing is specified left is assumed. (https://www.mathworks.com/help/matlab/creating_plots/add-text-to-specific-points-on-graph.html)

For example, lets add labels to the points in the diamond:

```
>> X = [1,0,-1,0,1]
>> Y = [0,1,0,-1,0]
>> plot(X, Y)
>> axis( [-2,2,-1,1] )
>> text(1,0,'(1,0)')
```

1.6 Some Basic Looping

Looping allow us to go through a list one element at a time and perform an action for each element in the list. The basic syntax is

```
for var = list
    Do Something
    Do Something Else
    Etc
end
```

Notice that we have dropped the >>. While loops work in the command window they are much easier to write up as scripts. From here on out we will assume script notation.

For example, we could square all of the numbers from 1 to 10:

```
for i = 1:10
    i^2
end
```

For another example, we could plot $\sin(x + n)$ where $n = 0, 1, 2, \ldots, 20$ on the same plot using `hold on`:

```
X = 0:.01:30
hold on
for n = 0:20
    plot(X, sin(X + n))
end
```

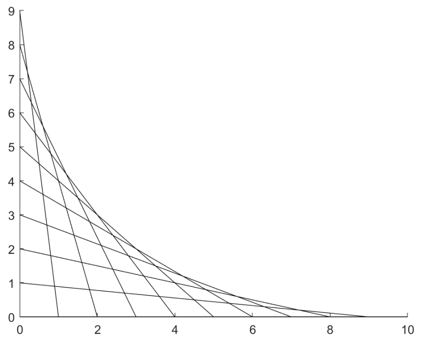
Finally, for our diamond above we can label the axes by accessing each of the four coordinates. MATLAB allows us to concatenate stings together using `+`, so `"Help" + "Me"` would yield `"HelpMe"`. For each of the coordinates `X(i)`, `Y(i)`, we can construct the label by `"(" + X(i) + "," + Y(i) + ")"`. For example, if `X(i)` is 4 and `Y(i)` is 3, then this becomes `"(4,3)"`. The final code looks like

```
X = [1,0,-1,0,1]
Y = [0,1,0,-1,0]
plot(X, Y)
axis( [-2,2,-1,1] )

for i = 1:4
    label = "(" + X(i) + "," + Y(i) + ")"
    text(X(i),Y(i),label)
end
```

Question Modify your grid code to add a coordinate label to each point of the grid.

Question Use a for loop and `hold on` to draw the lines connecting $(0, n)$ to $(10 - n, 0)$ for $n = 0, 1, \ldots, 10$ as below.



2 Solving Systems Of Equations Using MATLAB

We want to use MATLAB to solve systems of linear equations. Let us recall the standard setup. Consider a system of k linear equations in n unknowns

$$a_{11}x_1 + \ldots + a_{1n}x_n = b_1 \tag{1}$$
$$\vdots \tag{2}$$
$$a_{k1}x_1 + \ldots + a_{kn}x_n = b_k \tag{3}$$

In the standard matrix notation, we would write this equation as $Ax = b$, where

$$A = \begin{pmatrix} a_{11} & \ldots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{k1} & \ldots & a_{kn} \end{pmatrix}, \qquad x = \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix}, \qquad b = \begin{pmatrix} b_1 \\ \vdots \\ b_k \end{pmatrix}.$$

Example: Invertible Matrix Concretely, consider the following system of linear equations:

$$2x_1 - 2x_2 = 1\,, \tag{4}$$
$$x_1 + 4x_2 = 3\,. \tag{5}$$

In MATLAB, we can solve the equation using basic linear algebra and inverting the matrix:

```
>> A = [ 2, -2; 1, 4]
>> b = [1;3]
>> x = inv(A)*b
```

Here, `inv(A)` is A^{-1}. There are two reasons that this is not the correct way to do this computation in MATLAB:

1. If A is not square (or is square but not invertible) then the matrix isn't invertible.
2. Inverting matrices is slower than using Gaussian elimination, so this method is not application to large systems of equations.

Instead, we should put the augmented matrix [A | x] in reduced row echelon form using the function `rref()`.

```
>> Aug = [A, b]
>> rref(Aug)

ans =

    1.0000         0    1.0000
         0    1.0000    0.5000
```

Recall what this means in terms of the system of linear equations. By putting the augmented matrix in reduced row echelon form we have shown the original system is equivalent to

$$x_1 = 1\,, \tag{6}$$
$$x_2 = .5\,. \tag{7}$$

Equivalently, MATLAB uses the matrix division notation `A\b`, returning the result of the Gaussian elimination.

```
>> A\b

ans =

    1.0000
    0.5000
```

Example: Under-determined Systems Consider the following under determined system:

$$2x_1 - 2x_2 + 4x_3 = 1\,, \tag{8}$$
$$x_1 + 4x_2 - x_3 = 3\,. \tag{9}$$

We have chosen this system to have more than one solution. Lets put the augmented matrix into reduced row echelon form.

```
>> A = [2, -2, 4; 1, 4, -1]
>> b = [1;3]
>> rref([A, b])
```

```
ans =

    1.0000         0    1.4000    1.0000
         0    1.0000   -0.6000    0.5000
```

The Gaussian elimination gives us a unique way to write the solution space

$$x = \begin{pmatrix} 1 \\ .5 \\ 0 \end{pmatrix} + t \begin{pmatrix} -1.4 \\ .6 \\ 1 \end{pmatrix}, \qquad \text{for } t \in \mathbb{R}.$$

Contrast this to the result of using the shorthand

```
>> A\b

ans =

         0
    0.9286
    0.7143
```

The shorthand returns a single solution, normalized so that the first n vectors are 0. The Gaussian elimination solution is more complete, in particular the `A\b` solution does not indicate that the system is under determined. However we can get the null space using the function `null(A)`:

```
>> null(A)

ans =

   -0.7683
    0.3293
    0.5488
```

MATLAB naturally normalizes the null space to have unit length `[-0.7683, 0.3293, 0.5488] = 0.5488*[-1.4,.6,1]`.

Example: Over-determined Systems Now, lets look at the following over determined system:

$$2x_1 - 2x_2 = 1\,, \tag{10}$$
$$x_1 + 4x_2 = 3\,, \tag{11}$$
$$4x_1 - 7x_2 = 2\,. \tag{12}$$

Putting the system in reduced row echelon form yields

```
>> A = [2, -2; 1, 4; 4, -7]
>> b = [1;3;2]
>> rref([A, b])
```

```
ans =

     1     0     0
     0     1     0
     0     0     1
```

The reduced row echelon form indicates to us immediately that there is no solution, the last line indicating 0=1. If we try to use the shorthand `A\b` MATLAB will return an answer

```
>> A\b

ans =

    1.1805
    0.4211
```

This is MATLAB's best approximation of a solution.

Now, consider the system

$$2x_1 - 2x_2 = 1\,, \tag{13}$$
$$x_1 + 4x_2 = 3\,, \tag{14}$$
$$3x_1 + 2x_2 = 2\,. \tag{15}$$

Computing the reduced row echelon form we have

```
>> A = [2, -2; 1, 4; 3, -2]
>> b = [1;3;2]
>> rref([A, b])

ans =

    1.0000         0    1.0000
         0    1.0000    0.5000
         0         0         0
```

The solution space is then a single point `[1,.5]`. Alternatively, we can find a single solution and the null space:

```
>> A\b
>> null(A)

ans =

    1.0000
    0.5000

ans =

  2×0 empty double matrix
```

The result of the second computation is telling us that there is no null space, so there is only a single solution.

2.1 Problems: Linear Systems Of Mixed Rank

Use MATLAB to find the full solution space of the following equations

(a)

$$\begin{aligned} x_1 - x_2 + 2x_3 &= 1\,, && (16)\\ 2x_1 - 2x_2 + 4x_3 &= 1\,, && (17)\\ -3x_1 + 3x_2 - 6x_3 &= 1\,. && (18) \end{aligned}$$

(b)

$$\begin{aligned} x_1 - x_2 + 2x_3 &= 1\,, && (19)\\ x_1 - 4x_2 + x_3 &= -1\,, && (20)\\ 3x_1 + 3x_2 - 2x_3 &= 2\,. && (21) \end{aligned}$$

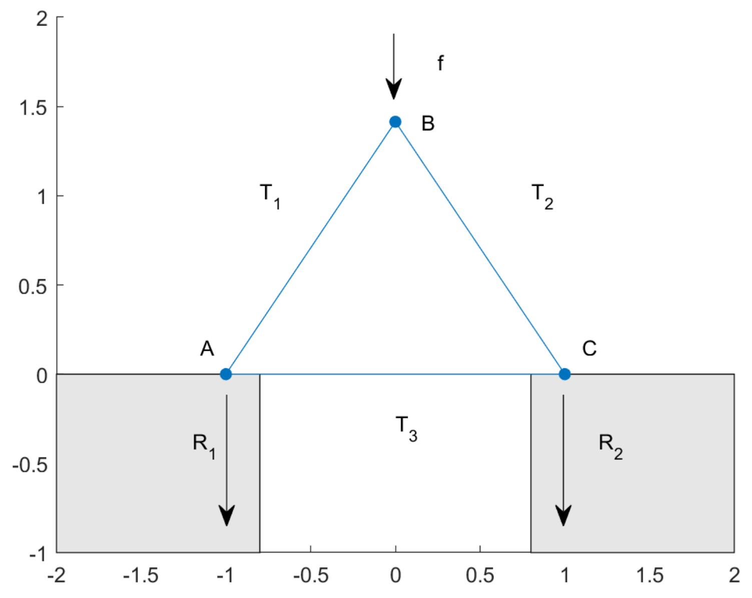
(c)

$$\begin{aligned} x_1 - x_2 + 2x_3 &= 1\,, && (22)\\ 4x_1 - 2x_2 + x_3 &= 1\,, && (23)\\ 2x_1 - 3x_3 &= -1\,. && (24) \end{aligned}$$

3 Linear Problem With Unknown Variables

Often linear systems depend not on constants but on parameters that are either measured from the outside world, or that must be determined by computation. For example, let us consider a simple example of economic computation.

In statics and construction physics, support placement and force allocation in trusses is a problem of linear algebra. Consider the 2D truss structure with vertices at (-1,0), (1,0) and (1,$\sqrt{2}$):

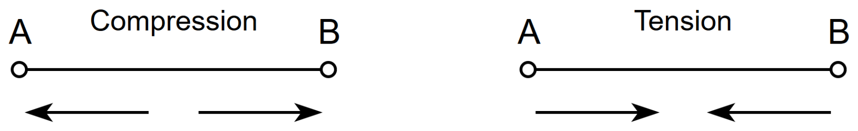


In the diagram above, the points of connection are called **verticies** and the lines connecting them represent **trusses**. In the **free body diagram** above, there are three kinds of forces working:

- $\vec{f}$ - The downward force of gravity and pressure on the structure.
- $\vec{R}_i$ - The upward forces of the ground on the structure.
- $\vec{T}_j$ - The forces passing through each truss.

Each force is a 2 dimensional vector. We will denote the **magnitude** of each force by it's unvectored symbol, that is $f := ||\vec{f}||$, $R_i = ||\vec{R}_i||$ and $T_j = ||\vec{T}_j||$.

Inner-truss Forces The force being exerted by each truss depends on weather the truss is in **tension** or **compression**. If the truss is compressed, it exerts a force against *both* of the connected vertices, pushing them apart. If the truss is in tension, it exerts a force pulling both vertices together.



When we solve a problem, **we will always start out assuming that all forces are in tension**, that way any force that ends up negative will be in compression.

Principles of Static Analysis Since the system is **static** (not moving) all of these forces must cancel out. Practically, the following must hold:

- The **net force** on the entire structure, that is the sum of all external forces, must be the 0 vector. That is $\vec{f} + \vec{R}_1 + \vec{R}_2 = \vec{0}$.

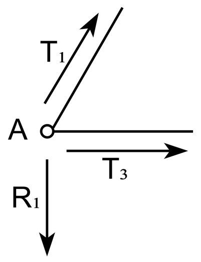
- Since the net force is $\vec{0}$, the x component and y component of the total force must also be 0, that is $\vec{f}_x + (\vec{R}_1)_x + (\vec{R}_2)_x = 0$.

In addition, none of the vertexes are moving, so * At each vertex A, B and C, the net force must be 0. For example, at A we must have $\vec{T}_1 + \vec{T}_3 + \vec{R}_1 = \vec{0}$. * Since the net force at each vertex is $\vec{0}$, both the x component and the y component of each force is 0 at each vertex.

3.0.1 Example: Computing Net Force At A Point

We will construct a linear system by write equations enforcing the static force balancing at each vertex. Remember that we assume that all inner-truss forces are **tension**, with the view to correct any negative forces to compression.

Force Equations at A:

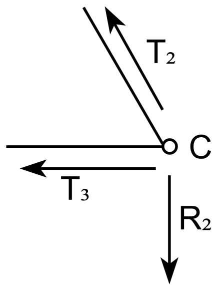


Lets look at vertex A. The **vertical force** on A will be the sum of the y components, $(\vec{R}_1)_y + (\vec{T}_1)_y = 0$. By convention, we will always add forces as if their arrow points out of the vertex, using the angle from the horizontal axis. The components of $\vec{T}_1 = (T_1 \cos \pi/3, T_1 \sin \pi/3)$ so we have the following vertical and horizontal forces

$$T_3 + (0.5)T_1 = 0, \qquad \text{Horiz.} \qquad (25)$$
$$-R_1 + (0.8660)T_1 = 0, \qquad \text{Vert.} \qquad (26)$$

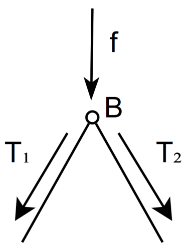
Similarly, we can find the force equations at B and C:

Force Equations at C:



$$-T_3 - (0.5)T_2 = 0, \qquad \text{Horiz.} \qquad (27)$$
$$-R_2 + (0.8660)T_2 = 0, \qquad \text{Vert.} \qquad (28)$$

Force Equations at B:



Finally, we want to treat f as a variable so we will write the equations around B with f on the right hand size.

$$-(0.5)T_1 + (0.5)T_2 = 0, \qquad \text{Horiz.} \tag{29}$$
$$-(0.8660)T_1 - (0.8660)T_3 = f, \qquad \text{Vert.} \tag{30}$$

A Linear System: Putting all of these equations together, we get the following linear system:

$$\begin{array}{ccccccc}
 & & (0.5)T_1 & & +T_3 & = & 0 \\
-R_1 & & +(0.8660)T_1 & & & = & 0 \\
 & & & -(0.5)T_2 & -T_3 & = & 0 \\
 & -R_2 & & +(0.8660)T_2 & & = & 0 \\
 & & -(0.8660)T_1 & -(0.8660)T_2 & & = & f \\
 & & -(0.5)T_1 & +0.5)T_2 & & = & 0
\end{array}$$

In matrix form this becomes

$$\begin{pmatrix} 0 & 0 & 0.5 & 0 & 1 \\ -1 & 0 & 0.8660 & 0 & 0 \\ 0 & 0 & 0 & -0.5 & -1 \\ 0 & -1 & 0 & 0.8660 & 0 \\ 0 & 0 & -0.8660 & -0.8660 & 0 \\ 0 & 0 & -0.5 & 0.5 & 0 \end{pmatrix} \begin{pmatrix} R_1 \\ R_2 \\ T_1 \\ T_2 \\ T_3 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ f \\ 0 \end{pmatrix}$$

In a script, you can separate the definition a matrix on different lines. Lets define varaibles `c=cos(pi/3)` and `s=sin(pi/3)` to make the matrix construction cleaner:

```
A = [ 0, 0, c, 0, 1;
     -1, 0, s, 0, 0;
      0, 0, 0,-c,-1;
      0,-1, 0, s, 0;
      0, 0,-s,-s, 0;
      0, 0,-c, c, 0]
```

Example: *Assume that a force of 1000 N is applied downward on vertex B. What is the horizontal force on exerted by truss T_3? Is it a compressing force or a tension force?*

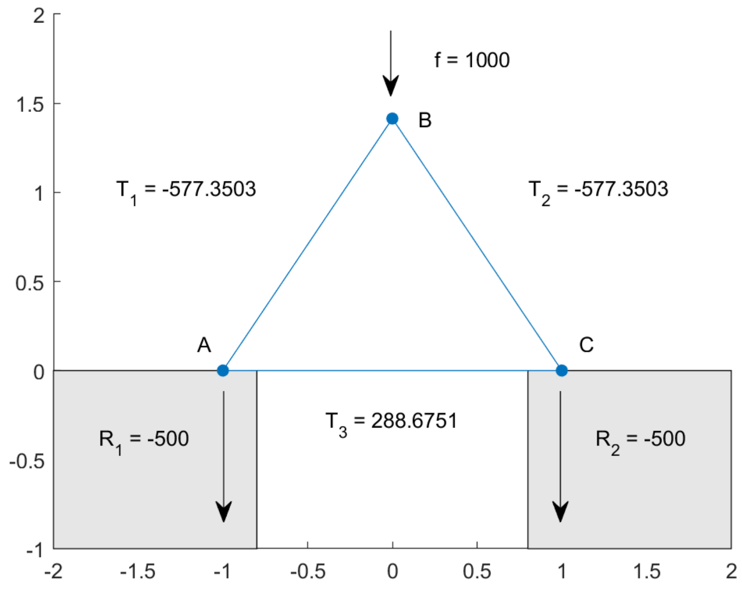
We just need to define the `B` vector for $f = 1000$ and ten perform the matrix division. We expect the system to have a single solution, we expect that $R_1 = R_2$ and that $T_1 = T_2$:

```
B = [0;0;0;0;1000;0]
A\B

ans =

 -500.0000
 -500.0000
 -577.3503
 -577.3503
  288.6751
```

If you get different results check you code. Since T_1 and T_2 are *negative*, both trusses carry compressive forces. In addition, we see that R_1 and R_2 act in the opposite direction to our free body diagram and indeed we would expect them to exert an upward force. We can annotate the free body diagram with these forces:



Example: *What does the force f have to be for the force on T_3 to be 1000 N?*

There are two ways to solve this problem that don't include trial and error: The simplest is to remember that the dependence of the forces is linear, so if we scale f by 1000/288.6751 we will have the required solution.

However, assume that you didn't remember that f was linear, of if the dependence on f was more complicated. MATLAB can create a **symbolic variable** that doesn't take on a strict value, just like a mathematical variable. We define f to be symbolic by using `syms f`:

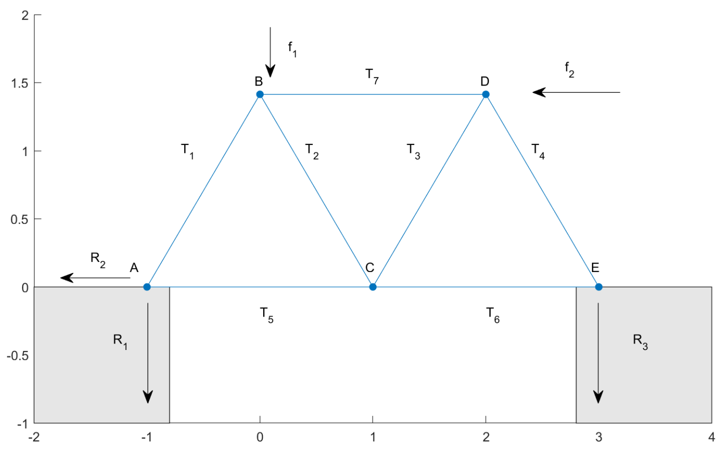
```
syms f
B = [0;0;0;0;f;0]
A\B

ans =
          -f/2
          -f/2
 -(3^(1/2)*f)/3
 -(3^(1/2)*f)/3
  (3^(1/2)*f)/6
```

Notice that this gives us the exact coefficient of proportionality between f and T_3, and that $\sqrt{3}/6 \approx 0.2887 \approx 288.6751/1000$.

3.1 Problem:

Consider the following truss structure, where all internal angles are $\pi/3$ radians.



1. Construct and solve the system of equations for the truss when $f_1 = 100$ and $f_2 = 1000$.
2. Assume $f_1 = 100$. What must f_2 be for the force T_3 to have magnitude 1000?
3. Assume the maximum force that can be applied along T_3 is 1000. Give an equation relating the forces f_1 and f_2 when $T_3 = 1000$.