

# An Introduction to MATLAB

for ACM/IDS 104: Applied Linear Algebra by Pavlos Stavrinos & Konstantin Zuev

## Profile

### What is MATLAB?

MATLAB is short for MATrix LABoratory. It is a numerical computation environment which provides a suite of tools for computation, visualization, and more. MATLAB excels at numerical computation.

### When should we use MATLAB?

- If we can afford to buy it, or can get it for free (like here at Caltech)
- For numerically intensive computations
- For plotting and dealing with data (analysis, visualization)
- For faster *development* than C and Fortran
- For faster *execution* than most other high level languages

### When should we *not* use MATLAB?

- When we have to pay for it
- For symbolic math, if we have access to Mathematica or Maple
- If we want to release code that does not require other users to have MATLAB

### Competitors:

- Mathematica and Maple [symbolic math packages]
- MATLAB imitations [FreeMat, Octave, SciLab, xmath]
- C and Fortran [staples of the high-performance community]
- C++, as well as C# etc. [modern Object-Oriented languages]
- Java and Python [also modern Object-Oriented languages]
- Perl, Bash and other shells [mainly data manipulation]
- S and R [statistics languages]

### MATLAB advantages:

- The language is intuitive and mathematically expressive → MATLAB is rather fast to learn
- The HELP menu is fantastic
- MATLAB is an industry standard (much like Microsoft Office) → the web is full of MATLAB resources
- MATLAB matrix manipulation algorithms (esp. for sparse matrices) are state of the art

### MATLAB disadvantages:

- Complex tasks (especially ones which require for loops). MATLAB can sometimes be slower than hand-coded C or Fortran
- Expensive

# Getting started

## What is this file?

This is an instance of a *MATLAB livescript* (`.mlx`). Livescripts can be thought of as the MATLAB equivalent for Python Jupyter Notebooks; they are interactive scripts where we can have text, LaTeX, code and figures all in one place. Furthermore, livescripts offer useful features such as auto-completion and help with debugging. Overall, Livescripts allow us to be more creative, organized and present our work in a more professional manner.

## Traditional m-files

Traditional *m-files* (`.m`) contain sequences of commands that are executed in order. They are very similar to using the command line, but much more convenient.

## The MATLAB desktop

- Command window, command prompt (command line interpreter)
- History window (history features are also available at the command prompt: <up> key, tab completion, drag-n-drop)
- Workspace, variable editor (and plot window)
- Current directory and file details editor
- The current folder selector, which helps to determine the search path (make sure the correct directory is selected. One way to do this is to right-click on the file tab and select the correct directory)
- The editor

All of the above windows can be rearranged as desired (Home → Layout)

## Commenting

Comments are a great way to document your code and make it more intuitive. In MATLAB, there are two ways to write comments:

```
% This is a single line comment

%{
This is a block comment.
Anything inside the braces is automatically
set to be a comment and is ignored by MATLAB
NB: we cannot have ANYTHING on the line that starts with "%{"
%}

%{
Comments are, by default, set to be green but
we can change this if desired
%}
```

## The HELP system

`help` shows the format(s) for using a command and directs to related commands. Without any arguments, it gives a hyperlinked list of topics to find help on. With a topic as an argument, it gives a list of subtopics. For example:

```
help lu
```

```
lu      lu factorization.
[L,U] = lu(A) returns an upper triangular matrix in U and a permuted
lower triangular matrix in L, such that  $A = L*U$ . The input matrix A can
be full or sparse.

[L,U,P] = lu(A) returns unit lower triangular matrix L, upper
triangular matrix U, and permutation matrix P such that  $P*A = L*U$ .

[L,U,P] = lu(A, outputForm) returns P in the form specified by
outputForm:
    'matrix' - (default) P is a matrix such that  $P*A = L*U$ .
    'vector' - P is a vector such that  $A(P,:) = L*U$ .
```

The following syntaxes *only* apply for a sparse input matrix A:

```
[L,U,P,Q] = lu(A) returns unit lower triangular matrix L, upper
triangular matrix U, and row/column permutation matrices P and Q such
that  $P*A*Q = L*U$ . For sparse matrices this is significantly more time
and memory efficient than the three-output syntax.
```

```
[L,U,P,Q,D] = lu(A) also returns a diagonal scaling matrix D such that
 $P*(D\backslash A)*Q = L*U$  for sparse A. Typically, the row-scaling leads to a
sparser and more stable factorization. Note that this is the
factorization used by sparse MLDIVIDE.
```

```
[___] = lu(A,THRESH) specifies the pivoting strategy employed by lu.
THRESH is a scalar or two-element vector with values in [0 1].
Increasing the value of THRESH tends to lead to higher accuracy, but
typically at a greater cost in time and memory. Depending on the number
of output arguments specified, the default value and requirements for
the thresh input are different:
```

- \* If 3 or fewer outputs are specified, then THRESH is a scalar. The default value is 1.0.
- \* If 4 or more outputs are specified, then THRESH is a two-element vector. The default value is [0.1 0.001].

```
[___] = lu(___, outputForm) specifies the output form of P and Q. Use
this option to return P and Q as vectors instead of matrices. P and Q
satisfy different identities depending on the number of outputs
specified and whether they are matrices or vectors:
```

With 4 outputs:

- 'matrix' - (default) P is a matrix such that  $P*A*Q = L*U$ .
- 'vector' - P is a vector such that  $A(P,Q) = L*U$ .

With 5 outputs:

- 'matrix' - (default) P is a matrix such that  $P*(D\backslash A)*Q = L*U$ .
- 'vector' - P is a vector such that  $D(:,P)\backslash A(:,Q) = L*U$ .

See also `qr`, `chol`, `ilu`, `mldivide`, `decomposition`, `inv`, `cond`.

Reference page for `lu`  
Other functions named `lu`

`doc` is like `help`, except it comes up in a different window, and can include more details.

```
%doc lu
```

`lookfor` is used when we do not know what command we want; it performs a keyword search through the documentation.

### lookfor Gauss

<code>quadgk</code>	- Numerically evaluate integral, adaptive Gauss-Kronrod quadrature.
<code>Gauss3Kronrod7</code>	- Gauss-Kronrod (3,7) pair.
<code>Gauss7Kronrod15</code>	- Gauss-Kronrod (7,15) pair.
<code>lqgreg</code>	- Form linear-quadratic-Gaussian (LQG) regulator
<code>lqgtrack</code>	- Forms a Linear-Quadratic-Gaussian (LQG) servo controller
<code>hwv</code>	- Create a Hull-White/Vasicek mean-reverting Gaussian diffusion model
<code>LinearGaussian2F</code>	- Create a 2 factor additive Gaussian interest rate model
<code>mutationgaussian</code>	- Gaussian mutation.
<code>imbilatfilt</code>	- Bilateral filtering of images with Gaussian kernels
<code>imgaussfilt</code>	- 2-D Gaussian filtering of images
<code>imgaussfilt3</code>	- 3-D Gaussian filtering of 3-D images
<code>imgaussfilt</code>	- 2-D Gaussian filtering of images
<code>imgaussfilt3</code>	- 3-D Gaussian filtering of 3-D images
<code>predopt</code>	- Executes the Predictive Controller Approximation based on Gauss Newton.
<code>firgauss</code>	- FIR Gaussian digital filter design.
<code>gauspuls</code>	- Gaussian-modulated sinusoidal pulse generator.
<code>gaussdesign</code>	- Gaussian FIR Pulse-Shaping Filter Design
<code>gaussfir</code>	- Gaussian FIR Pulse-Shaping Filter Design.
<code>gausswin</code>	- Gaussian window.
<code>gmonopuls</code>	- Gaussian monopulse generator.
<code>gaussian</code>	- Construct a Gaussian pulse shaping filter designer.
<code>psgaussnsym</code>	- PSGAUSSIANSYM Construct an PSGAUSSIANSYM object.
<code>gausswin</code>	- WINDOWRCOS Construct a gausswin object
<code>tffunc</code>	- time and frequency domain versions of a cosine modulated Gaussian pulse.
<code>fitrgp</code>	- Fit a Gaussian Process Regression (GPR) model.
<code>RegressionGP</code>	- Gaussian Process Regression (GPR) model.
<code>fitgmdist</code>	- Fit a Gaussian mixture distribution to data.
<code>cdf</code>	- CDF for the Gaussian mixture distribution.
<code>cluster</code>	- GMDISTRIBUTION/CLUSTER Cluster data for Gaussian mixture distribution.
<code>disp</code>	- Display a Gaussian mixture distribution object.
<code>display</code>	- Display a Gaussian mixture distribution object.
<code>gmdistribution</code>	- Gaussian mixture distribution class.
<code>pdf</code>	- PDF for a Gaussian mixture distribution.
<code>estep</code>	- E-STEP for Gaussian mixture distribution
<code>gmcluster</code>	- Gaussian mixture fit.
<code>gmrnd</code>	- Random vectors from a multivariate Gaussian mixture model.
<code>addinv</code>	- Add the inverse Gaussian distribution.
<code>hypergeom</code>	- Gauss' Hypergeometric function

`demo` gives video guides and example code in a new window

```
%demo
```

## Crawling

*Before writing your MATLAB code, it is always good practice to get rid of any leftover variables and figures from previous scripts.*

```
clc; clear; close all;
```

## Hello World

As programming tradition (and meme culture) dictates, we shall write a "Hello World" command to properly induct ourselves into MATLAB code:

```
disp("Hello World :)");
```

```
Hello World :)
```

## MATLAB as a calculator

Let us perform some elementary calculations and observe MATLAB's behavior:

```
4 + 6 % displays result
```

```
ans = 10
```

```
1 + 2; % suppresses display of result (but the calculation is done)
```

```
3 + 97 % white space doesn't matter where it shouldn't
```

```
ans = 100
```

```
% Wrapping lines
```

```
3 + ...
```

```
5
```

```
ans = 8
```

```
%{  
As long as i or j have not been assigned to as regular variables,  
MATLAB treats them as sqrt(-1)  
%}  
i^2
```

```
ans = -1
```

```
j^2
```

```
ans = -1
```

```
exp(pi*i)
```

```
ans = -1.0000 + 0.0000i
```

```
% If we set  
i = 1;  
% then i is 1 now:  
i + 1
```

```
ans = 2
```

```
% but if we clear its value:  
clear i
```

```
% i becomes sqrt(-1) again
disp(i);
```

```
0.0000 + 1.0000i
```

## Variables

MATLAB does not require us to set variables, and is case sensitive:

```
x = 9823475269456;
%disp(X);
```

`disp(X)` returns an error because `X` does not match `x`. Variable names in MATLAB must start with an upper or lower case letter and can *only* be followed by letters, digits and underscores. Variable names cannot be the same as built-in MATLAB keywords. (`iskeyword` can be used to check this). Now, let us see how we can set and use variables:

```
a = 2;
b = 2;
a^b
```

```
ans = 4
```

```
% MATLAB stores the result of the last calculation as a variable named 'ans'
ans + 1
```

```
ans = 5
```

```
x = 8; a + x % the semicolon suppresses output AND allows
```

```
ans = 10
```

```
% to add a second command.
%{
who and whos give information on the variables currently defined.
We can also see this information in the 'workspace' tab, which
lists all the global data we have in memory
%}
who
```

Your variables are:

```
a    ans  b    x
```

```
whos
```

Name	Size	Bytes	Class	Attributes
a	1x1	8	double	
ans	1x1	8	double	
b	1x1	8	double	
x	1x1	8	double	

## Vectors and Matrices

Matrices are the fundamental MATLAB datatype. It is essential we know how to work with them:

```
clc; clear;  
x = [1, 2, 3] % row vector: separate elements in the same row with commas
```

```
x = 1×3  
    1     2     3
```

```
y = [1 2 3] % row vector: spaces also work to separate elements in a row
```

```
y = 1×3  
    1     2     3
```

```
z = [1; 2; 3] % column vector: separate elements in the same column with ;
```

```
z = 3×1  
     1  
     2  
     3
```

```
z = 1:20 % can use the range notation to generate row vectors
```

```
z = 1×20  
    1     2     3     4     5     6     7     8     9    10    11    12    13 ...
```

```
z = 1:2:20 % the "2" means that we go from 1 to 20 in increments of 2
```

```
z = 1×10  
    1     3     5     7     9    11    13    15    17    19
```

```
z = 20:-2:0
```

```
z = 1×11  
    20    18    16    14    12    10     8     6     4     2     0
```

```
z = [0:.1:1] % the array brackets are optional
```

```
z = 1×11  
    0    0.1000    0.2000    0.3000    0.4000    0.5000    0.6000    0.7000 ...
```

```
% a 3-by-3 matrix:
```

```
A = [1, 2, 3; 4, 5, 6; 7, 8, 9]
```

```
A = 3×3  
     1     2     3  
     4     5     6  
     7     8     9
```

```
% the spaces and commas separate columns  
% and the semicolons separate rows
```

```
% A*x is invalid! Just like in linear algebra, quantities need to have  
% the correct dimensions to make a valid MATLAB expression  
x*A % this works: row vector times matrix is a row vector
```

```
ans = 1×3  
    30    36    42
```

```
A*x' % x' is a (conjugate) transposition of x (.' for real transpose)
```

```
ans = 3x1
    14
    32
    50
```

```
% Some operations are "matrix" operations, and others are "component-wise"
% aka "element-wise".
```

```
B=A*A % this is matrix multiplication
```

```
B = 3x3
    30    36    42
    66    81    96
   102   126   150
```

```
C=A.*A % this works component-wise, c_ij=(a_ij)*(a_ij)
```

```
C = 3x3
     1     4     9
    16    25    36
    49    64    81
```

```
B=A^2 % this is matrix multiplication, e.g. "A*A"
```

```
B = 3x3
    30    36    42
    66    81    96
   102   126   150
```

```
C=A.^A % this is element-wise, e.g. "A.*A"
```

```
C = 3x3
         1         4         27
       256       3125      46656
    823543   16777216  387420489
```

```
B=exp(A) % this is element-wise
```

```
B = 3x3
103 x
    0.0027    0.0074    0.0201
    0.0546    0.1484    0.4034
    1.0966    2.9810    8.1031
```

```
C=expm(A) % this is an inherent matrix operation
```

```
C = 3x3
106 x
    1.1189    1.3748    1.6307
    2.5339    3.1134    3.6929
    3.9489    4.8520    5.7552
```



```
% use A' to take the conjugate transpose of a matrix, and .' to take the  
% real transpose; e.g.:
```

```
A = [0 i; -i, 0]
```

```
A = 2x2 complex  
    0.0000 + 0.0000i    0.0000 + 1.0000i  
    0.0000 - 1.0000i    0.0000 + 0.0000i
```

```
A.'
```

```
ans = 2x2 complex  
    0.0000 + 0.0000i    0.0000 - 1.0000i  
    0.0000 + 1.0000i    0.0000 + 0.0000i
```

```
A'
```

```
ans = 2x2 complex  
    0.0000 + 0.0000i    0.0000 + 1.0000i  
    0.0000 - 1.0000i    0.0000 + 0.0000i
```

```
% You can convert row (column) vectors to column (row) vectors using ' or .'  
x = [1, 5, 6, 7, 7]'
```

```
x = 5x1  
    1  
    5  
    6  
    7  
    7
```

## First Steps

### More on Matrices

Practically every task in MATLAB uses matrices in some way. Let us explore some more ways to work with them:

```
A = [1 4 7; 2 5 8; 3 6 9]
```

```
A = 3x3  
    1    4    7  
    2    5    8  
    3    6    9
```

```
% Indices:  
% use [ ] to create a matrix  
% use ( ) to access an element
```

```
% Reading  
A(1,1) %indexing is 1-based in MATLAB
```

```
ans = 1
```

```
A(1,2)
```

```
ans = 4
```

```
A([1,3],1)
```

```
ans = 2x1
     1
     3
```

```
A(2,[2,3])
```

```
ans = 1x2
     5     8
```

```
A(1:3,1)
```

```
ans = 3x1
     1
     2
     3
```

```
A(:,1)
```

```
ans = 3x1
     1
     2
     3
```

```
A(1,:)
```

```
ans = 1x3
     1     4     7
```

```
A([1,3,2],:) % permute rows
```

```
ans = 3x3
     1     4     7
     3     6     9
     2     5     8
```

```
A([1 3 2],[1 3 2]) % permute rows and columns
```

```
ans = 3x3
     1     7     4
     3     9     6
     2     8     5
```

```
% Getting the size
length([1 2 3]) %only for row/column vectors
```

```
ans = 3
```

```
[m,n] = size(A)
```

```
m = 3
n = 3
```

```
% Writing
A(1,1) = 3
```

```
A = 3x3
    3    4    7
    2    5    8
    3    6    9
```

```
A(1:3,2:3) = NaN    % NaN stands for "Not a Number"
```

```
A = 3x3
    3    NaN    NaN
    2    NaN    NaN
    3    NaN    NaN
```

```
A(1:2,1:3) = [100 200 300; 400 500 600]
```

```
A = 3x3
   100    200    300
   400    500    600
    3     NaN     NaN
```

```
% Linear indexing
% MATLAB uses "column major order", meaning a matrix is stored
% as a linear array, one column after another
A = [1 4 7; 2 5 8; 3 6 9]
```

```
A = 3x3
    1    4    7
    2    5    8
    3    6    9
```

```
A(:)
```

```
ans = 9x1
    1
    2
    3
    4
    5
    6
    7
    8
    9
```

```
A(1:4:end) % handy trick to extract the diagonal:
```

```
ans = 1x3
    1    5    9
```

```
% if A is an N*N, then, A(1:N+1:end)
% the special keyword "end" refers to the last entry
```

```
% Reshaping
reshape(1:9,3,3)    % reshapes array
```

```
ans = 3x3
    1    4    7
    2    5    8
    3    6    9
```

```
B = [1 2 3 4; 5 6 7 8]
```

```
B = 2x4
    1     2     3     4
    5     6     7     8
```

```
reshape(B,4,[]) % with 4 rows
```

```
ans = 4x2
    1     3
    5     7
    2     4
    6     8
```

```
reshape(B,[],4) % with 4 columns
```

```
ans = 2x4
    1     2     3     4
    5     6     7     8
```

```
reshape(B,4,2) % 4-by-2
```

```
ans = 4x2
    1     3
    5     7
    2     4
    6     8
```

```
% Higher dimensional arrays
```

```
M = reshape(1:27,3,3,3)
```

```
M =
M(:,:,1) =
    1     4     7
    2     5     8
    3     6     9
```

```
M(:,:,2) =
   10    13    16
   11    14    17
   12    15    18
```

```
M(:,:,3) =
   19    22    25
   20    23    26
   21    24    27
```

```
M = repmat((1:3)',2,2) % repeats copies of a matrix
```

```
M = 6x2
    1     1
    2     2
    3     3
    1     1
    2     2
    3     3
```

```
% 'block' operations ("generalization" of repmat)
```

```
A = [1 2; 3 4]
```

```
A = 2x2
```

```
1    2  
3    4
```

```
B = [A, 2*A; -A, 2*A]
```

```
B = 4x4
```

```
1    2    2    4  
3    4    6    8  
-1   -2    2    4  
-3   -4    6    8
```

```
% Generating standard matrices
```

```
eye(5) % the 'eye' identity
```

```
ans = 5x5
```

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

```
ones(5) % the same as ones(5,5)
```

```
ans = 5x5
```

```
1    1    1    1    1  
1    1    1    1    1  
1    1    1    1    1  
1    1    1    1    1  
1    1    1    1    1
```

```
ones(5,3)
```

```
ans = 5x3
```

```
1    1    1  
1    1    1  
1    1    1  
1    1    1  
1    1    1
```

```
rand(2,2)
```

```
ans = 2x2
```

```
0.8147    0.1270  
0.9058    0.9134
```

```
randn(3,3)
```

```
ans = 3x3
```

```
0.3188    0.3426   -1.3499  
-1.3077    3.5784    3.0349  
-0.4336    2.7694    0.7254
```

```
zeros(4,4)
```

```
ans = 4x4
```

```

0    0    0    0
0    0    0    0
0    0    0    0
0    0    0    0

```

```
inf(3,3)
```

```

ans = 3x3
    Inf    Inf    Inf
    Inf    Inf    Inf
    Inf    Inf    Inf

```

```
nan(4,1)
```

```

ans = 4x1
    NaN
    NaN
    NaN
    NaN

```

```

% rank, determinant, and inverse:
A=eye(3,3);
rank(A)

```

```
ans = 3
```

```
det(A)
```

```
ans = 1
```

```
inv(A)
```

```

ans = 3x3
    1    0    0
    0    1    0
    0    0    1

```

## Solving Linear Systems $Ax = b$

When trying to solve a linear system of the form  $Ax = b$ , the "backslash" operator `\` is your best friend. This is a built in MATLAB feature that works as follows:

```

%{
Inputs: A, b
if A is square:
    if A is triangular:
        Use forward or backward substitution to solve for x in O(n^2)
    else if A is symmetric positive definite:
        Use Cholesky solver to solve for x in O(n^3)
    else:
        Use LU decomposition to solve for x in O(n^3)
    end if
else:
    Obtain least squares solution x
end if
return x
}

```

```
%}
```

You do not need to worry about the exact details of how "backslash" works. Just, keep in mind that there exist *three* cases:

1. **no solution:**  $\backslash$  finds a least squares solution.
2. **unique solution:**  $\backslash$  finds the exact solution.
3. **infinitely many solutions:**  $\backslash$  finds a particular solution.

The general solution to the system  $Ax = b$  describes all possible solutions. The general solution can be obtained as follows:

### 1) Check consistency:

- if  $\text{rank}(A) == \text{rank}([A,b]) \implies$  solution exists (1 or  $\infty$  many)
- if  $\text{rank}([A,b]) > \text{rank}(A) \implies$  no solution # least squares solution

```
% Example: overdetermined system, m>n.
```

```
clc; clear;  
A=rand(11,10);  
b=rand(11,1);  
rank(A)
```

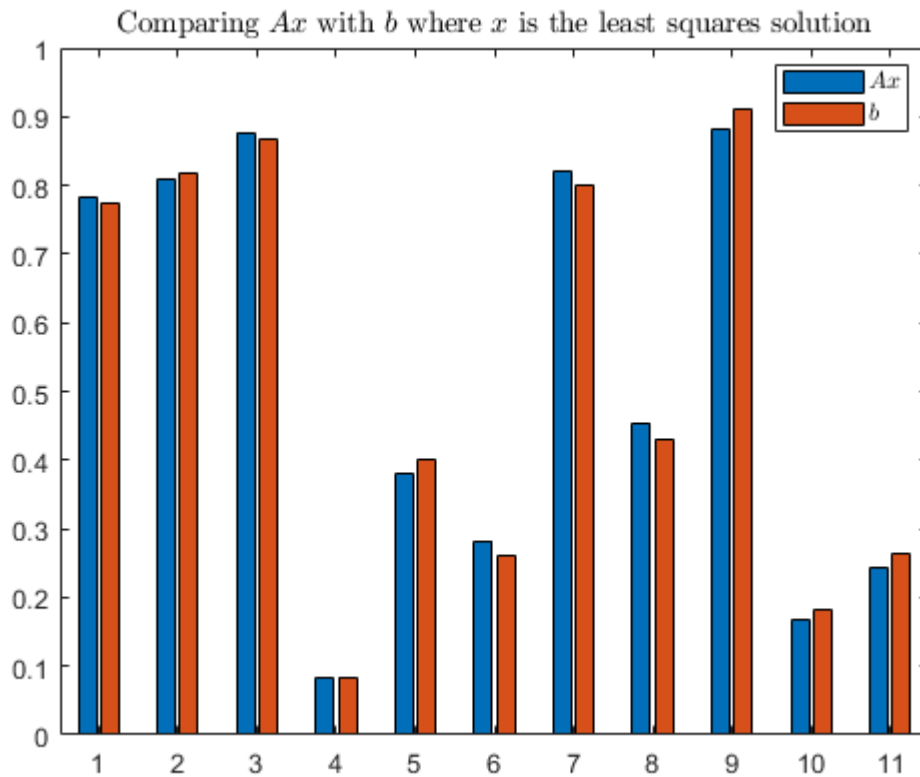
```
ans = 10
```

```
rank([A,b])
```

```
ans = 11
```

```
x=A\b    % least squares solution
```

```
figure;  
bar([A*x,b]); % A*x does not equal to b, but the two vectors are close.  
legend("$Ax$", "$b$", "Interpreter", "latex", "Location", "best");  
title("Comparing $Ax$ with $b$ where $x$ is the least squares solution", "Interpreter", "latex");
```



```
norm(A*x-b) % check error
```

```
ans = 0.0595
```

Now, suppose that the system is consistent, i.e.  $\text{rank}(A) == \text{rank}([A,b])$ :

## 2) Check uniqueness:

- if  $\text{rank}(A) == n$ , where  $[~,n] = \text{size}(A) \implies$  solution is unique and  $x = A \backslash b$
- if  $\text{rank}(A) < n \implies$  there exist  $\infty$  many solutions

```
% Example: unique solution
```

```
A=rand(10,10);
```

```
b=rand(10,1);
```

```
rank(A)
```

```
ans = 10
```

```
rank([A,b])
```

```
ans = 10
```

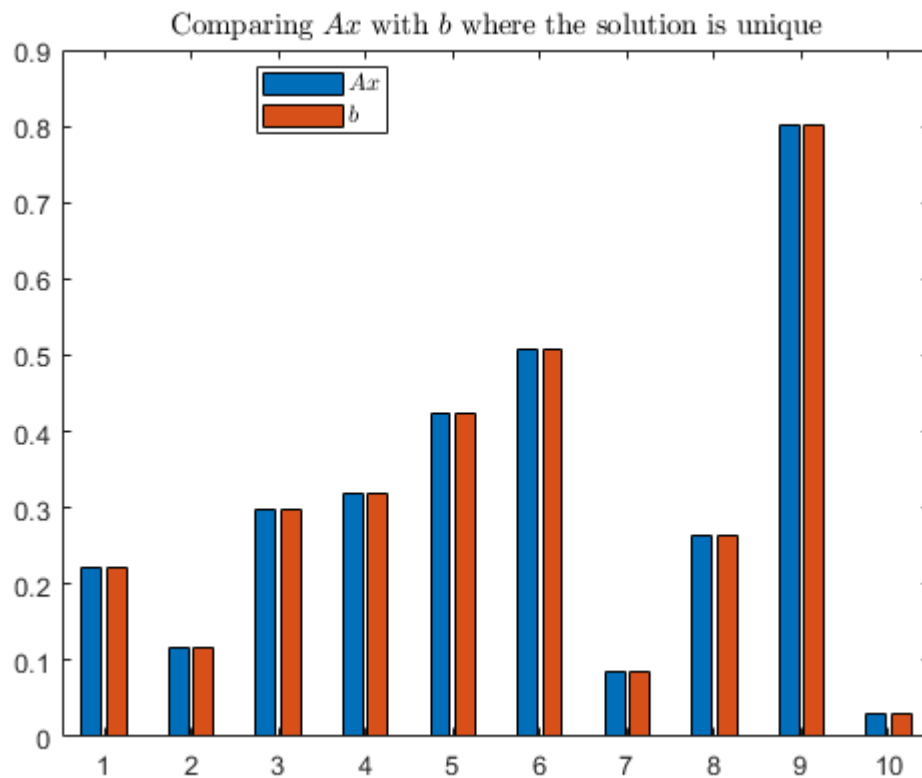
```
[~,n] = size(A)
```

```
n = 10
```

```
x=A\b
```



```
figure;
bar([A*x,b]);
legend("$Ax$", "$b$", "Interpreter", "latex","Location", "best");
title("Comparing $Ax$ with $b$ where the solution is unique","Interpreter","latex");
```



```
norm(A*x-b) % check error
```

```
ans = 3.1225e-16
```

Now, suppose that the solution is not unique:

### 3) Construct solution

The general solution of  $Ax = b$  is the sum of a particular solution  $x_0$  of  $Ax = b$  plus a linear combination of the basis vectors  $v_1, \dots, v_k$  ( $k = n - r$ ) for the solution space of the corresponding homogeneous system  $Ax = 0$ .

- $x_0 = A \backslash b$  gives particular solution of  $Ax = b$
- `null(A)` returns a basis for the solution space to  $Ax = 0$

```
% Example: undetermined system, m<n.
A=rand(5,10);
b=rand(5,1);
rank([A,b])
```

```
ans = 5
```

```
rank(A)
```

```
ans = 5
```

```
[~,n] = size(A)
```

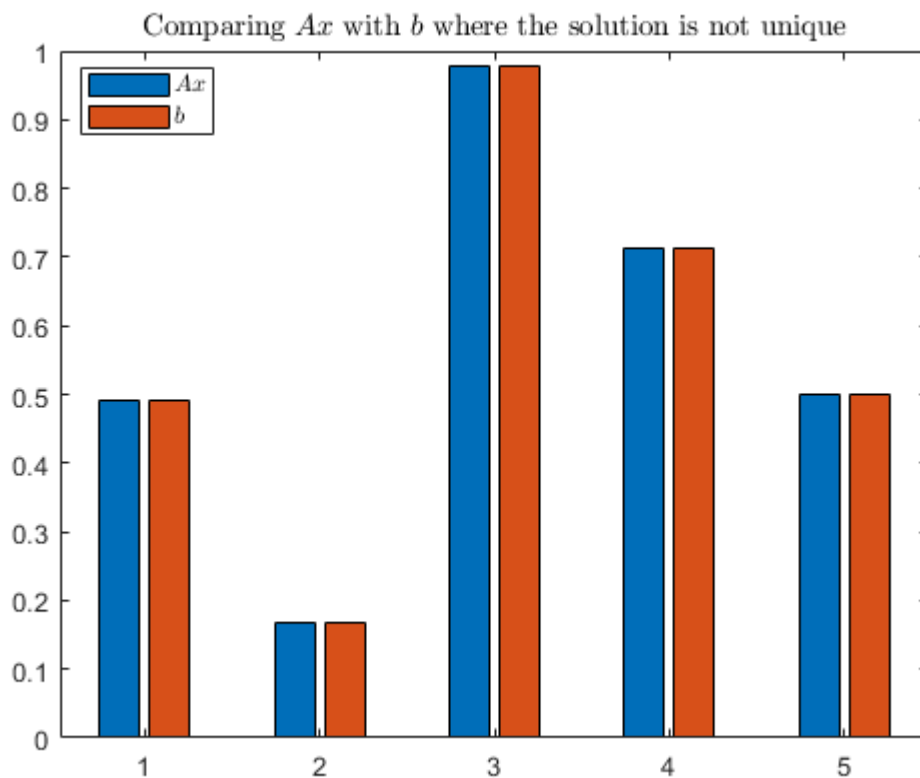
```
n = 10
```

```
x0=A\b
```

```
A*x0-b % check
```

```
V=null(A)
```

```
% construct solution  
x=x0+V(:,1)+V(:,2)+V(:,3)+V(:,4)+V(:,5);  
figure;  
bar([A*x,b]);  
legend("$Ax$", "$b$", "Interpreter", "latex", "Location", "best");  
title("Comparing $Ax$ with $b$ where the solution is not unique","Interpreter","latex");
```



```
norm(A*x-b) % check error
```

```
ans = 9.5827e-16
```

## Walking

## Comparing different solvers

In MATLAB, there exist several different functions that can be used to solve linear systems of the form  $Ax = b$ . Here, we will inspect how some of the most popular methods behave:

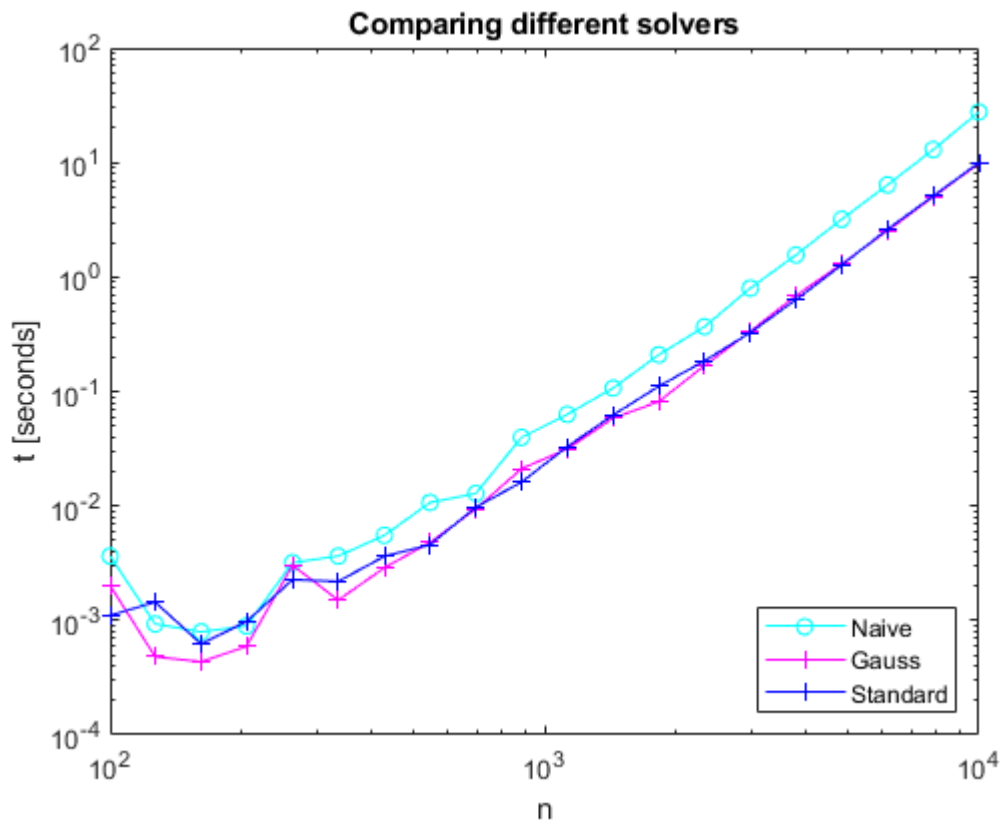
```
clear; clc;

% The naive way
naiveSolve = @(A,b) inv(A)*b;
[ts_naive,ns_naive] = generateTiming(naiveSolve); % measures performance
% ns_naive = sizes of test systems
% ts_naive = time (in sec) required for solving

% Using Gaussian Elimination
% Gauss.m
[ts_Gauss,ns_Gauss] = generateTiming(@Gauss);

% The standard way: "\"
% The backslash operator attempts to decide what solution method
% will work well. It should be your first choice most of the time.
standardSolve = @(A,b) A\b;
[ts_std,ns_std] = generateTiming(standardSolve);

figure;
loglog(ns_naive,ts_naive,'co-',ns_Gauss,ts_Gauss,'m+- ',ns_std,ts_std,'b+- ');
xlabel('n');
ylabel('t [seconds]');
legend('Naive','Gauss','Standard','Location','southeast');
title('Comparing different solvers');
```

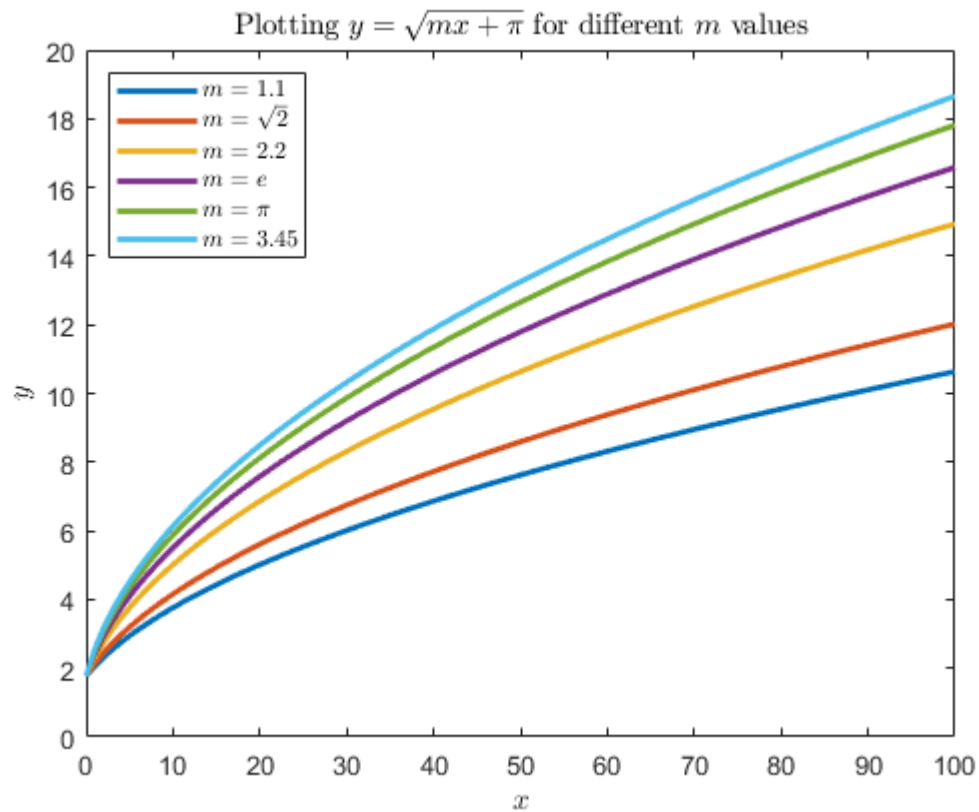


## Figures & Loops

Visualization is extremely important in MATLAB. Creating figures allows us to bring our code to life and present our results in a compelling and professional manner. In this section we will review a variety of ways to create figures in MATLAB, and we will see how loops work and can assist us.

To start, let us plot, in a single figure, the line  $y = \sqrt{mx + c}$  on  $x \in [0, 100]$  for  $c = \pi$  and different values of  $m$ :

```
clear; clc;
c = pi; % fixed value of c
m = [1.1 sqrt(2) 2.2 exp(1) pi 3.45]; % choice of m values
x = linspace(0, 100); % x values
figure; % <- ALWAYS tell MATLAB you are starting a new figure
for i = 1 : length(m)
    y = sqrt(m(i) * x + c); % line equation
    plot(x, y, "LineWidth", 2); % plotting
    hold on % <- this tells MATLAB to plot everything in one figure
end
% LABELS
title("Plotting  $y = \sqrt{mx + \pi}$  for different  $m$  values", "Interpreter","latex");
ylabel(" $y$ ", "Interpreter","latex");
xlabel(" $x$ ", "Interpreter","latex");
legend(" $m = 1.1$ ", " $m = \sqrt{2}$ ", " $m = 2.2$ ", " $m = e$ ", " $m = \pi$ ", " $m = 3.45$ ", "Interpreter", "Interpreter", "Interpreter", "Interpreter", "Interpreter", "Interpreter")
```

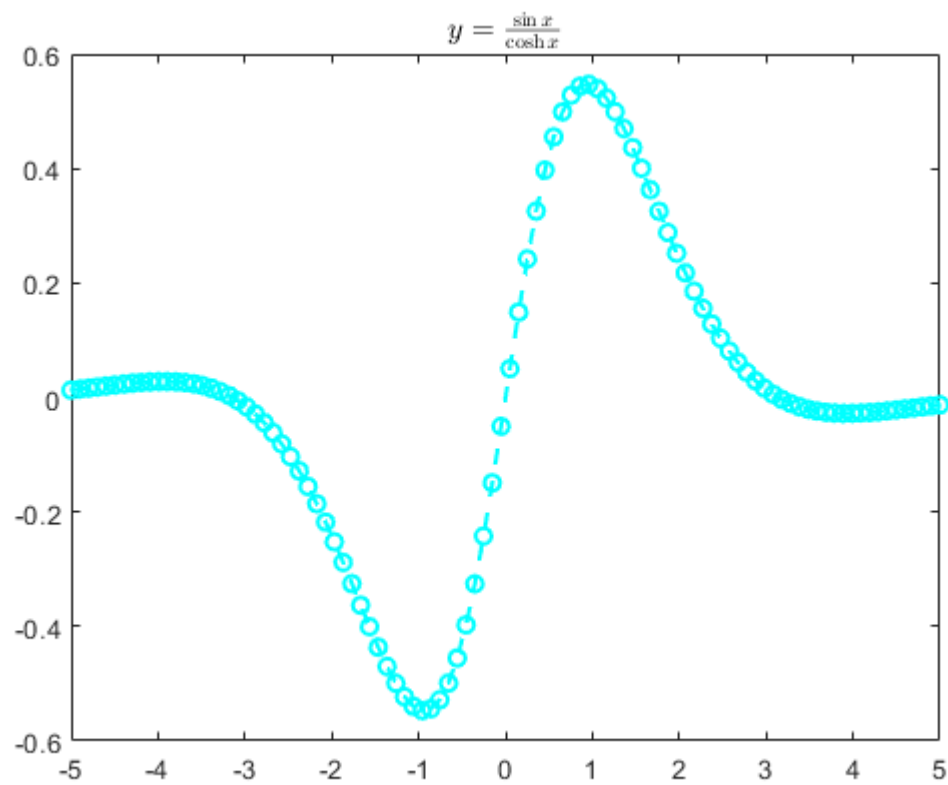


Now, what if we want to plot a function, say:

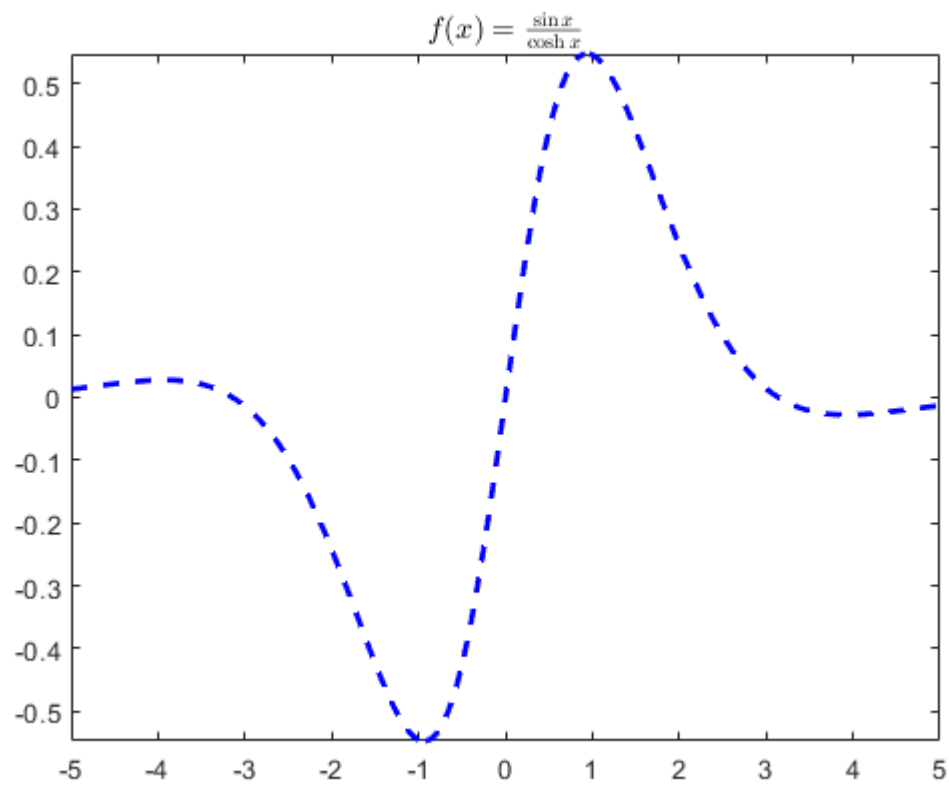
$$f(x) = \frac{\sin x}{\cosh x}, \quad x \in [-5, 5]$$

There are a few ways to do this:

```
% Using plot()
clear; clc;
x = linspace(-5, 5);
y = sin(x) ./ cosh(x);
figure;
% We can specify different linestyles and linewidths
plot(x,y,"c--o","LineWidth",1.5);
title("$y = \frac{\sin x}{\cosh x}$","Interpreter","latex");
```

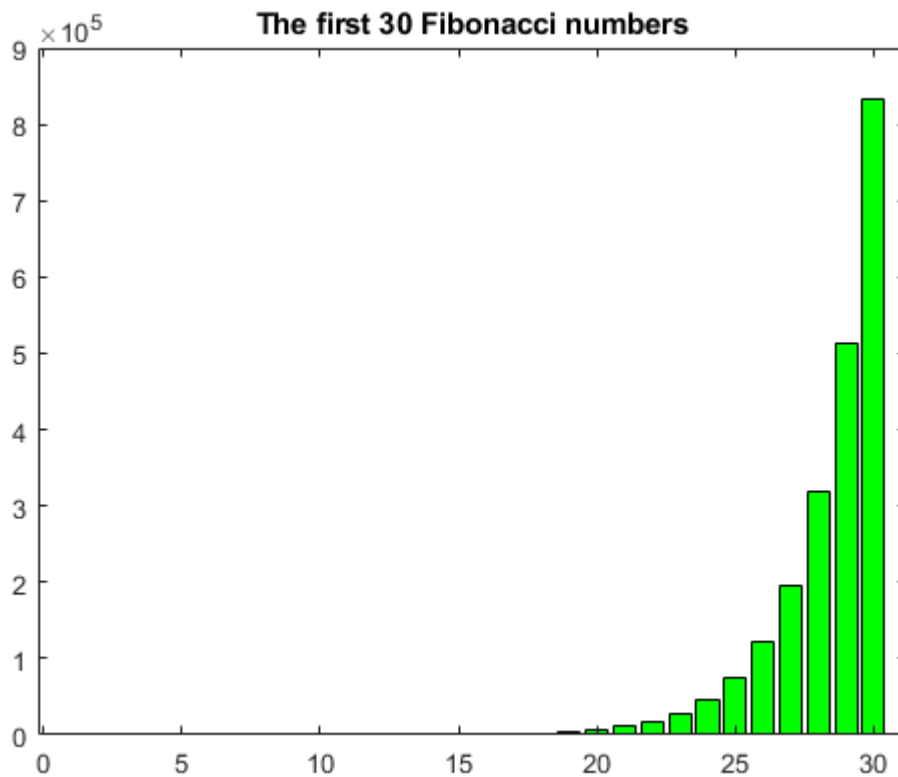


```
% Using fplot
f = @(x) sin(x) ./ cosh(x); % we can define functions like this
figure;
fplot(f,[-5, 5],"b--","LineWidth",2);
title("$f(x) = \frac{\sin x}{\cosh x}$","Interpreter","latex");
```



We can generate the first 30 Fibonacci numbers, using the built in `fibonacci()`, and plot them on a bar graph to see how fast they grow:

```
figure;  
bar(fibonacci(1:30), "g");  
title("The first 30 Fibonacci numbers");
```



Moving on, let us see how we can use `subplot()` to plot four interesting 3D functions:

```
clc; clear;
% Let us define our functions
f = @(x, y) cos(sqrt(x.^2+y.^2)).*exp(-sqrt(x.^2+y.^2)/5);
g = @(x, y) sin(10*(x.^2+y.^2))./10;
h = @(x, y) sin(5*x).*cos(5*y)/5;
r = @(u,v) 2 + sin(7.*u + 5.*v);
funx = @(u,v) r(u,v).*cos(u).*sin(v);
funy = @(u,v) r(u,v).*sin(u).*sin(v);
funz = @(u,v) r(u,v).*cos(v);

figure;
subplot(2,2,1);
fsurf(f, [-10 10 -10 10]);
xlabel("$x$", "Interpreter","latex");
ylabel("$y$", "Interpreter","latex");
zlabel("$z$", "Interpreter","latex");

subplot(2,2,2);
fsurf(g, [-1 1 -1 1]);
xlabel("$x$", "Interpreter","latex");
ylabel("$y$", "Interpreter","latex");
zlabel("$z$", "Interpreter","latex");

subplot(2,2,3);
fsurf(h, [-1 1 -1 1]);
xlabel("$x$", "Interpreter","latex");
```

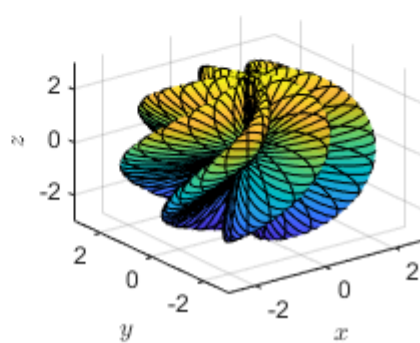
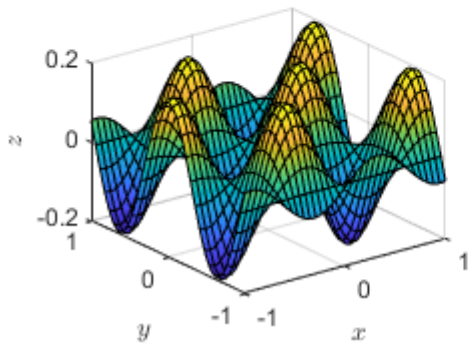
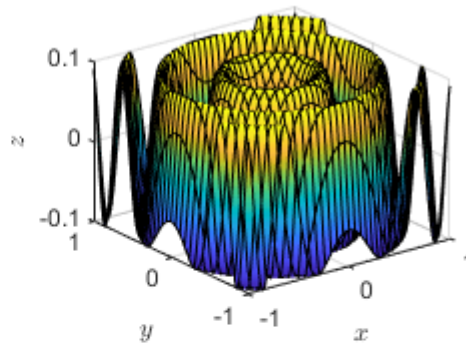
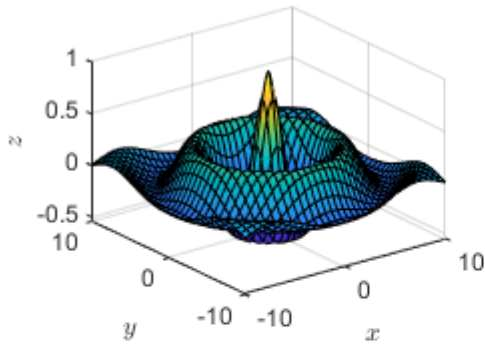


```

ylabel("$y$", "Interpreter","latex");
zlabel("$z$", "Interpreter","latex");

subplot(2,2,4);
fsurf(funx,funy,funz,[0 2*pi 0 pi]);
xlabel("$x$", "Interpreter","latex");
ylabel("$y$", "Interpreter","latex");
zlabel("$z$", "Interpreter","latex");

```



% you can click and drag the subplots around

These are just a few examples of the endless plotting possibilities in MATLAB! Now, let us focus more on loops. Suppose we want to find the first integer  $n$  for which  $n!$  is a 166-digit number:

```

clear; clc;
n = 1;
n_factorial = 1;
while n_factorial < 1e166
    n = n + 1;
    n_factorial = n_factorial * n;
end
disp(n); % just like the number for this class :)

```

Now, let us code one big nested loop to create a matrix with 0 on the main diagonal, 104 on the adjacent diagonals, and -9 everywhere else:

```
clc; clear;
rows = 9;
cols = 9;
A = ones(rows,cols);
for i = 1 : cols
    for j = 1 : rows
        if j == i
            A(j,i) = 0;
        elseif abs(j-i) == 1
            A(j,i) = 104;
        else
            A(j,i) = -9;
        end
    end
end
disp(A);
```

```

    0   104    -9    -9    -9    -9    -9    -9    -9
  104     0   104    -9    -9    -9    -9    -9    -9
   -9   104     0   104    -9    -9    -9    -9    -9
   -9    -9   104     0   104    -9    -9    -9    -9
   -9    -9    -9   104     0   104    -9    -9    -9
   -9    -9    -9    -9   104     0   104    -9    -9
   -9    -9    -9    -9    -9   104     0   104    -9
   -9    -9    -9    -9    -9    -9   104     0   104
   -9    -9    -9    -9    -9    -9    -9   104     0
```

## Running

### Sparse Matrices

In a few fields, the term *sparse* has become a buzzword. What does it mean? There's no quantitative definition, but the qualitative definition is easy: a vector or matrix is *sparse* when it has many zero entries. In MATLAB, there are two ways to store a matrix. The first way, which we've been using so far, is to store all entries. If we know the dimensions of the matrix, and have a convention of how the entries are stored, then there is no need to store the indices. The second method, which MATLAB calls the *sparse* datatype, is to only store the nonzero entries. But, with this scheme, we need to store the indices also, hence there's a penalty in memory. It is very important to realize that MATLAB's definition of *sparse* only refers to the way a matrix is stored. The same matrix can be sparse, or non-sparse (aka "full").

Here is an example:

```
clc; clear;
A = randn(2)           % Method 1 of storing a matrix, aka "full" storage
```

```
S = sparse(A)          % Method 2 of storing a matrix, aka "sparse" storage
```

```
S =
    (1,1)    -1.2112
```

```
(2,1)    -0.1194
(1,2)    -0.1303
(2,2)    1.1001
```

```
% The "sparse" function converts a matrix to the sparse storage format. We
% can go the other direction using the "full" command, e.g.
% B = full(S);
whos
```

Name	Size	Bytes	Class	Attributes
A	2x2	32	double	
S	2x2	88	double	sparse

Now, suppose we have a matrix that really is sparse. Then, the sparse matrix format has the advantage:

```
clear; A = zeros(100);
A(45,60) = 1; % method 1
S = sparse(A) % method 2
```

```
S =
  (45,60)      1
```

```
whos
```

Name	Size	Bytes	Class	Attributes
A	100x100	80000	double	
S	100x100	824	double	sparse

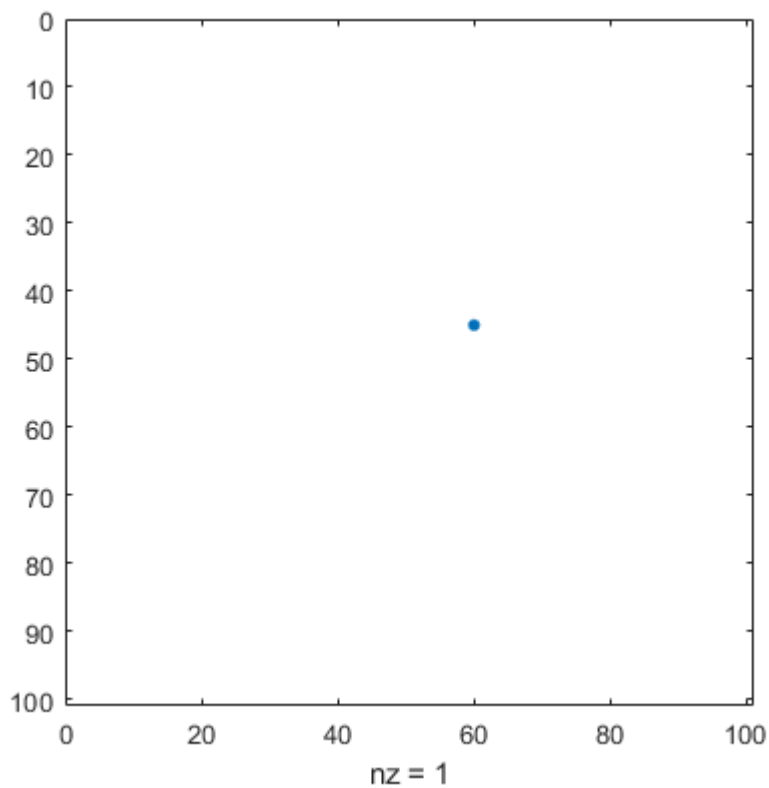
We can create a sparse matrix on its own, i.e. we don't have to convert from a full matrix first. The simplest method is the following:

```
clear;
S = sparse(100,100); % makes an all zero 100 x 100 sparse matrix
% Remark: S = sparse(100) is NOT what you want.
% sparse(100) would make a 1x1 sparse matrix, with the (1,1) entry being 100.
```

```
S(45,60) = 4
```

```
S =
  (45,60)      4
```

```
figure;
spy(S) % very useful for seeing the sparsity pattern.
```



```
% sizes
[m,n]=size(S)
```

```
m = 100
n = 100
```

```
k=nnz(S) %number of non-zeros
```

```
k = 1
```

Generating random sparse matrices:

```
A=sprand(10,10,0.1) % random, 10-by-10, sparse matrix with approximately
```

```
A =
(6,1)    0.6164
(7,1)    0.9606
(9,1)    0.5869
(7,5)    0.1823
(9,5)    0.5781
(2,6)    0.1139
(1,7)    0.6460
(3,8)    0.3889
(7,8)    0.5254
(3,10)   0.6495
```

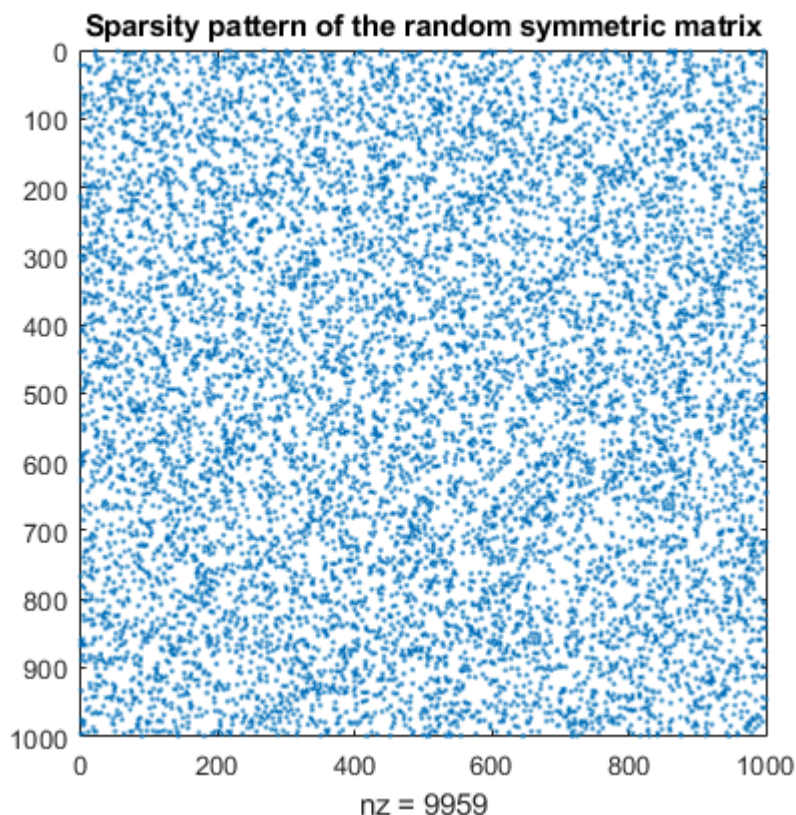
```
% 0.1*10*10 uniformly distributed nonzero entries
B=sparse(rand(4,4)) % converts a full matrix into sparse form by squeezing
```

```
B =
```

(1,1)	0.9804
(2,1)	0.2082
(3,1)	0.5527
(4,1)	0.3671
(1,2)	0.3639
(2,2)	0.4750
(3,2)	0.5634
(4,2)	0.3103
(1,3)	0.0761
(2,3)	0.4172
(3,3)	0.6023
(4,3)	0.6225
(1,4)	0.1873
(2,4)	0.8563
(3,4)	0.5049
(4,4)	0.0622

```
% out any zero elements. If a matrix contains many zeros, converting the
% matrix to sparse storage saves memory.
B = sprandsym(1000,.01); % this is a 1000 x 1000 sparse symmetric matrix
                        % 1% of its entries are randomly selected to
                        % be nonzero; on those entries, the value is
                        % chosen from a uniform distribution.
```

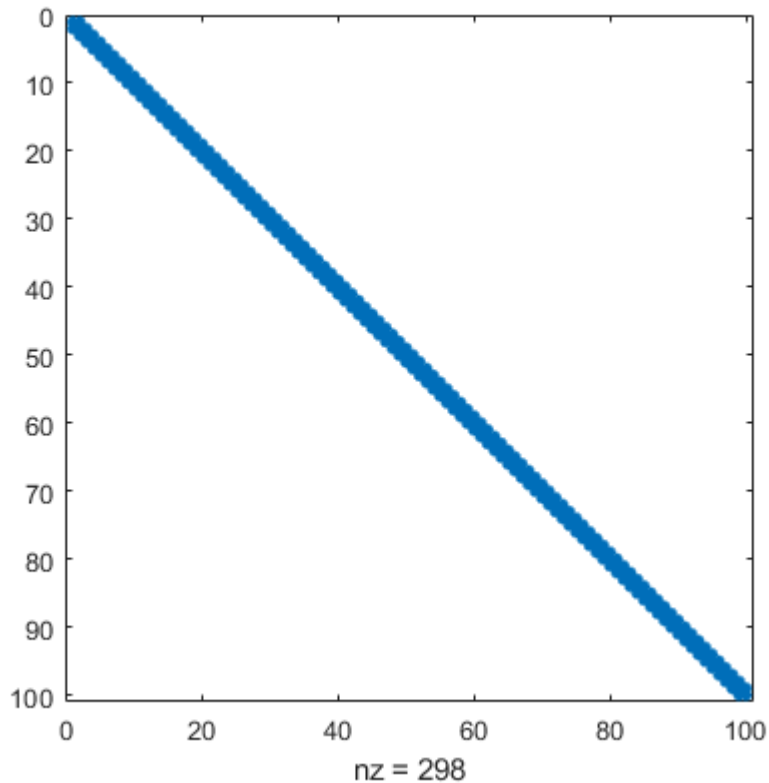
```
figure;
spy(B); title('Sparsity pattern of the random symmetric matrix');
```



This is more or less the way sparse matrices are constructed in practice:

```
n=100;
iSet = [2:n,1:n,1:n-1];
jSet = [1:n-1,1:n,2:n];
```

```
vals = [-ones(1,n-1),2*ones(1,n),-ones(1,n-1)];
A=sparse(iSet, jSet, vals, n, n);
figure;
spy(A)
```



```
% with spalloc
clear; clc;
n = 10^5; % matrix size
p = n*0.1; % number of non-zero elements
tic
S = spalloc(n,n,p); % allocates space for sparse n-by-n matrix
                    % with p nonzero entries.
%S=zeros(n,n);
for k=1:p
    i = ceil(n*rand); % ceil rounds toward positive infinity
    j = ceil(n*rand);
    S(i,j) = rand;
end
toc
```

Elapsed time is 0.304668 seconds.

This is a more standard way (avoiding loops) to accomplish the same task:

```
clear; clc;
n = 10^5;
p = n*0.1;
tic
```

```
S = sparse(...
    ceil(n*rand(1,p)),... % row indices
    ceil(n*rand(1,p)),... % col indices
    rand(1,p),...         % values
    n,n);                 % dimensions, nnz
toc
```

Elapsed time is 0.005679 seconds.

## Importance of memory pre-allocation

Memory pre-allocation is an important element that will help optimize your MATLAB code. Consider the following sequence:

$$x_{k+1} = \frac{1}{2} (x_k^3 + \sqrt{x_k} - 4), \quad x_1 = 2$$

Suppose we want to calculate the first  $10^6$  terms of this sequence and store them in a row vector:

### Part I - No pre-allocation

Here, we will initialize the final output as a scalar and append a new term to it in each round of the loop:

```
clc; clear;
lim = 10^6;
tic
out_one = 2; % scalar
for i = 1 : lim - 1
    out_one(i + 1) = 1/2 * (out_one(i)^3 + sqrt(out_one(i)) - 4); % next value
end
toc
```

Elapsed time is 2.034169 seconds.

```
disp(out_one(10^6)); % display last value
```

Inf

### Part II - With pre-allocation

Now, we will initialize the final output as an array with the desired size and update one term in the array in each round of the loop:

```
out_two = zeros(1, lim); % pre-allocating row vector
tic
out_two(1, 1) = 2; % setting the first value
for k = 1 : lim - 1
    out_two(1, k + 1) = 1/2 * (out_one(k)^3 + sqrt(out_one(k)) - 4); % next value
end
toc
```

Elapsed time is 1.907927 seconds.

```
disp(out_two(10^6)); % display last value
```

```
%{
The following code checks that every term in the two sequence vectors is
the same.
%}
if out_one == out_two
    disp("The two vectors are equal!")
end
```

```
The two vectors are equal!
```

As we can see, the code with pre-allocated memory runs faster! In your future assignments you should *always try preallocating the maximum amount of space* required for an array at first instead of continuously resizing the array.

## Functions and liveFunctions

Like most of its competitors, MATLAB allows us to define our own functions. This can be done in two different ways when using livescripts:

1. Have the function in a separate *m-file* (.m)
2. Create a Live Function in the same livescript

In our discussion above about comparing solvers, we used two functions defined in separate m-files (Gauss.m, generateTiming.m). If you open these files, you can view MATLAB's syntax for defining functions. Here are some important points to remember when using functions defined in separate m-files:

- Only *one* function can be defined per m-file (that's how MATLAB recognizes it is a function file)
- Ensure that the function and the script you are working with are *in the same directory*. Otherwise, MATLAB will not be able to import and use the function

The alternative way of defining a function is to create a Live Function. Live Functions can exist within livescripts, but they *must be located at the bottom of the script*. Here is an example:

```
clc; clear;
A = [13,12,24,54,65,48,23,65,87,23,87,12,65,98,12,65,3,12];
[m, v, sd] = simplestat(A);
G = gram(A);
disp(m);
```

```
42.6667
```

```
disp(v);
```

```
969.5294
```

```
disp(sd);
```

```
31.1373
```



```
disp(G);
```

Columns 1 through 12

169	156	312	702	845	624	299	845	1131
156	144	288	648	780	576	276	780	1044
312	288	576	1296	1560	1152	552	1560	2088
702	648	1296	2916	3510	2592	1242	3510	4698
845	780	1560	3510	4225	3120	1495	4225	5655
624	576	1152	2592	3120	2304	1104	3120	4176
299	276	552	1242	1495	1104	529	1495	2001
845	780	1560	3510	4225	3120	1495	4225	5655
1131	1044	2088	4698	5655	4176	2001	5655	7569
299	276	552	1242	1495	1104	529	1495	2001
1131	1044	2088	4698	5655	4176	2001	5655	7569
156	144	288	648	780	576	276	780	1044
845	780	1560	3510	4225	3120	1495	4225	5655
1274	1176	2352	5292	6370	4704	2254	6370	8526
156	144	288	648	780	576	276	780	1044
845	780	1560	3510	4225	3120	1495	4225	5655
39	36	72	162	195	144	69	195	261
156	144	288	648	780	576	276	780	1044

Columns 13 through 18

845	1274	156	845	39	156
780	1176	144	780	36	144
1560	2352	288	1560	72	288
3510	5292	648	3510	162	648
4225	6370	780	4225	195	780
3120	4704	576	3120	144	576
1495	2254	276	1495	69	276
4225	6370	780	4225	195	780
5655	8526	1044	5655	261	1044
1495	2254	276	1495	69	276
5655	8526	1044	5655	261	1044
780	1176	144	780	36	144
4225	6370	780	4225	195	780
6370	9604	1176	6370	294	1176
780	1176	144	780	36	144
4225	6370	780	4225	195	780
195	294	36	195	9	36
780	1176	144	780	36	144

```
function [m, v, sd] = simplestat(A)
%{
This is the MATLAB function syntax.
-> [m, v, sd] are the outputs of the function.
-> "simplestat" is the name of the function. (you can change that if
      you wish but make sure you change
      every function call as well!)
-> A is the argument of the function.
%}
m = mean(A);
v = var(A);
sd = sqrt(v);
end

function G = gram(A)
```

```
G = A'*A;  
end
```