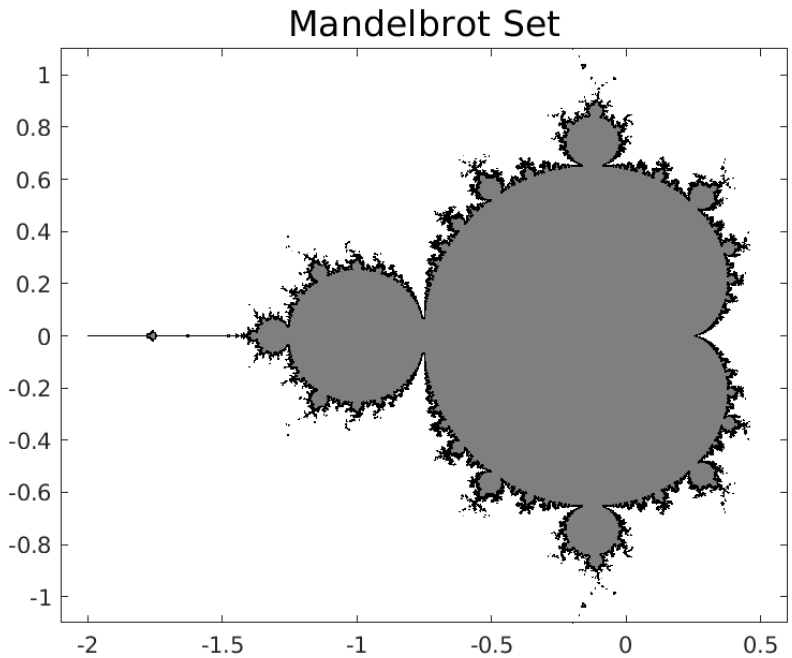
CS319: Scientific Computing (with MATLAB)

Niall Madden

Flow control; Matrices and Vectors

Week 3: **9am and 4pm**, 25 Jan 2023



Source: MATLAB Guide

Lab times

	Mon	Tue	Wed	Thu	Fri
9 – 10			LECTURE		
10 – 11			LAB 1		
11 – 12			LAB 2		
12 – 1					
1 – 2					
2 – 3					
3 – 4					
4 – 5			LECTURE		

- Attend either (or both) Labs 1 or 2.
- Can anyone attend neither?
- Any requests for another hour?

Bitbucket git repo

You should now have access to the CS319 git repository at https://bitbucket.org/niallmadden/2223-cs319/src. If not, check your email for an invitation, or send me an email.

Today, in CS319:

1. 1: Output/Input
   - Output
   - format
   - fprintf
   - Input
2. 2: Flow of control – `if`
3. 3: `while` loops
   - Example: Newton's method
4. 4: `for` loops
5. 5: Vectors and matrices
   - Common matrices
6. 6. Vectors
   - Accessing elements
   - Vector indexing
   - Useful functions
7. 7. Matrices
   - Vector indexing
   - Operations
   - Special matrices
   - Last example

Other reading:

- Chapter 1 of The MATLAB Guide:
  https://doi-org.nuigalway.idm.oclc.org/10.1137/1.9781611974669
- Chapter 3 of Learning MATLAB:
  https://doi-org.nuigalway.idm.oclc.org/10.1137/1.9780898717662

There are several different ways of getting output from MATLAB:

- When running a command, just omit the semi-colon from the line.
- Use the `disp()` function which outputs a single variable.
- The results of the above two can be controlled using the `format` function. Try

```
1    disp(pi)
     format long
3    disp(pi)
     format shortE
5    disp(pi)
```

There are other uses of the `format` instruction. Use `>> doc format` to read more.

However, the most useful may be:

```
format compact
```

Not as useful, but fun:

```
format rat
```

To reset:

```
format default
```

Mostly, we will use the `fprintf()` function, which is a little like a `f`-string in Python (and almost identical to `printf()` in C).

This is especially useful, because we can mix text and variable values, can specify how many decimal places, to output to, etc. Also, this is used to write to files.

Syntax:

- First argument is always a string (text that starts and ends with either a single or double quote).
- That string may contain a **conversion character**: `%` followed by a letter, e.g., `%f`
- The `%f` is replaced with the value of the second argument.
- Further conversion characters and arguments are allowed.

Common conversion characters:

`f`	fixed-point representation of a float.
`e` or `E`	exponent notation
`g` or `G`	let MATLAB guess if `f` or `e`
`c` or `s`	single character or string
`d` or `i`	integer is better.

You can also set the

- field width
- precision.

Examples:

```
1  fprintf('pi = %f\n', pi);     % pi = 3.141593
   fprintf('pi = %7.1f\n', pi); % pi =      3.1
3  fprintf('pi = %7.3f\n', pi)  % pi =    3.142
   fprintf('pi = %7.5f\n', pi)
5  fprintf('pi = %7.7f\n', pi)
   fprintf('pi = %7.6e\n', pi)
```

In that previous example, `\n` is an "escape character". It causes a newline to be printed.

Other escape characters:

- `\t`
- `\\`
- `%%`

Since MATLAB is an interactive system, reading input in a script is not very common. But if we must:

```
x = input('Tell me something: ')
```

```
1 x = input('Tell me something: ', 's')
```

2: Flow of control – `if`

`if` statements are used to conditionally execute part of your code.

Syntax: `if/else`:

```
if( exprn )
    statements to execute if exprn evaluates is true
else
    statements if exprn evaluates as 0
end
```

- The `else` statement is optional, but good practice.
- The `end` statement is needed.
- Indentation is good practice, but not required.

2: Flow of control – `if`

The argument to `if()` is a **logical expression**.

Example

- Equality: `x == 8` or `m == 'c'`
- Inequality: `y ~= x`
- Less than: `y < 1`
- Less than or equal to: `z <= pi`
- Greater than: `X > 9`
- Greater than or equal to: `q123 >= 1/2`

More complicated examples can be constructed using the operators

- **AND** `&&`
- **OR** `||`.

2: Flow of control – if

Eg01_EvenOdd.m

```
Number = input("Please enter an integrer: ");
if ( mod(Number,2) == 0)
   fprintf("%d is an even number.\n",  Number);
else
   fprintf("%d is an odd number.\n",  Number);
end
```

More complicated examples are possible:

Syntax: if/elseif/else:

```
if( exp1 )
    statements to execute if exp1 evaluates is true
elseif ( exp2 )
    statements run if exp1 is "false" but exp2 is "true"
else
    "catch all" statements if both exp1 and exp2 false.
end
```

2: Flow of control – if

Eg02_Grades.m

```
%% Eg02_Grades.m
%   Date   : Jan 2023
%   What   : Example of using if-elseif-else
NumberGrade = input("Please enter the grade (percent): ")
    ;
if ( NumberGrade >= 70 )
   LetterGrade = 'A';
elseif ( NumberGrade >= 60 )
   LetterGrade = 'B';
elseif ( NumberGrade >= 50 )
   LetterGrade = 'C';
elseif ( NumberGrade >= 40 )
   LetterGrade = 'D';
else
   LetterGrade = 'E';
end
fprintf("%2d%% corresponds to a %c grade\n", ...
   NumberGrade, LetterGrade);
```

2: Flow of control – `if`

The other main flow-of-control structure is
`switch / case / otherwise`

It has limited used (I find), since it doesn't involve any relational operators. But it can be helpful if you have set some parameter in your code.

Eg03_Switch.m

```
x = [12, 5, 59, 24];
plottype = 'bar'; % One of 'bar', 'pie', 'pie3'
switch plottype
    case 'bar'
        bar(x)
        title('Bar Graph')
    case {'pie','pie3'}
        pie3(x)
        title('Pie Chart')
    otherwise
        warning('Unexpected plot type. No plot created.')
end
```

3: `while` loops

A `loop` is a programming structure that allows for some piece of code to be repeated.

There are two main types of loop in MATLAB:

- `while`: preform a set of instructions as long as a given logical statement holds true;
- `for`: for each element in a vector, preform a set of instructions.

3: while loops

Syntax: while:

```
while( exp1 )
    statements to execute so long as exp1 evaluates is true
end
```

Eg04_Countdown_while.m

```
c = 10;
while (c>0)
    fprintf("%i... ", c);
    c=c-1;
end
fprintf("Zero!\n");
```

One of the most classic problem in scientific computing is solving nonlinear equations: given a function f, find x such that $f(x) = 0$.

And one of the most important methods for solving this is **Newton's Method:** if x_k is a good estimate for x, then

$$x_{k+1} = x_k - f(x_k)/f'(x_k),$$

To implement this method we need to know how many iterations to preform. Since we are trying to solve $f(x) = 0$, we can use $f(x_k)$ is a good measure for how good an estimate x_k is. That is, we iteration `while` $|f(x_k)|$ is greater than some chosen value.

We also need to know how to define functions in MATLAB. We'll study that in detail next week, but for now we just need to know that the syntax is:

```
>> fun_name = @(x)(formula for function in x)
```

The point term here is the use of the `@` symbol.

We can plot functions defined in this with using `fplot()`.

In the following example, we'll use Newton's method to solve

$$f(x) = x^2 - 2$$

for $x > 0$. That is, we are estimating $\sqrt{2}$.

Eg05Newton.m

```
f = @(x)x.^2-2;
df = @(x)2*x;
fplot(f, [0,3]);

xk = 1;
k = 0;
fprintf("k=%2d, xk=%f, f(xk)=%8.2e\n", ...
    k, xk, f(xk))
while (abs(f(xk)) > 1.0e-6)
    k=k+1;
    xk = xk - f(xk)/df(xk);
    fprintf("k=%2d, xk=%f, f(xk)=%8.2e\n", ...
        k,xk,f(xk))
end
```

4: `for` loops

A `for` loop is used when we want to

1. Repeat the execution of a block of code a fixed number of times; or
2. Execute a block of code for each element in a (row) vector.

These two applications are actually the same, but we'll treat them separately for now...

Syntax: for: fixed number of iterations

```
for i=1:N
    statements to executed N times
end
```

In the next example, we will use the `nthprime()` function to display the first 10 prime numbers.

Eg06Primes.m

```
%% File   : Eg05_Countdown.m
%  Date   : Jan 2023 (CS319 Week 03)
%  What   : Use a for loop to display 1st 10 primes
for i=1:10
   fprintf("The %2d-th prime is %2d\n", i, nthprime(i));
end
```

Here is a slightly more general version:

Loop over integers from a to b

```
for i=a:b
    // code to execute inside loop
    // First time, i=a
    // Next, i=a+1
    // ...
    // Last:  i=b
end
```

Explanation of $a : b$ and of $a : h : b$

4: for loops

Example: we'll re-do the `while` count-down example using a `for`-loop.

Eg07Countdown.m

```
4 for i=10:-1:1
    disp(i);
6 end
  disp(' Zero!');
```

4: `for` loops

1. In the most general use of `for`, the syntax is
   `for Index = ListOfValues`
2. `ListOfValues` is a row `vector`, that is, a $1 \times n$ matrix.
3. At each step through the loop, `Index` takes the next element of the list.
4. If the list is empty, nothing is done.
5. The instructions iterated are between for `for` and `end` lines.
6. Can also use `continue` or `break`, but it is rarely necessary.

5: Vectors and matrices

MATLAB stands for "matrix laboratory". The core goal of the original version of MATLAB was to be a "matrix calculator" (https://dl.acm.org/doi/10.1145/3386331

So working with matrices and vectors is simpler than in just about any other language.

In fact, if you assign a single number to a variable, it is stored as a 1×1 matrix.

Similarly, vectors are just

- $1 \times n$ matrix for a row vectors
- $n \times 1$ matrix for a column vectors

As well as these notes, you should read Chapter 3 of "Learning MATLAB": https://epubs-siam-org.nuigalway.idm.oclc.org/doi/pdf/10.1137/1.9780898717662.ch2

5: Vectors and matrices

The simplest way to define a matrix is to list its entries:

- List the entries between square brackets
- Place a space or comma between columns;
- Place a semicolon at the end of rows

```
>> A = [1, 2, 3; -1, -2, 0; 0, 1, 2]
A =

     1     2     3
    -1    -2     0
     0     1     2
```

5: Vectors and matrices

```
>> b = [5 5 5]  % commas are optional
b =
     5     5     5
```

```
>> c = [-1; -2; -3]
c =
    -1
    -2
    -3
```

Use `whos` to check the size of these arrays. You can also use the **size** function: `size(A)`

5: Vectors and matrices

It is easy to combine matrices and vectors to make larger ones. With the examples above, we could set

```
>> X=[A; b]
```

or

```
>> Y=[A, c]
```

What happens if you try?

```
>> X=[A, b] % note comma
```

or

```
>> Y=[A; c] % note semicolon
```

In the case of certain special or common matrices, there are functions to construct them:

- `I=eye(N)` makes the $N \times N$ identity matrix

```
>> I=eye(3)
I =
     1     0     0
     0     1     0
     0     0     1
```

- The zero matrix: `Z = zero(m,n)` sets Z to be an $m \times n$ matrix, all of whose entries are zero.

```
>> Z = zeros(1,4)
Z =
     0     0     0     0
```

- `ones(m,n)` returns the $m \times n$ matrix, all of whose entries are 1.

```
>> ones(3,1)
ans =
     1
     1
     1
```

- Random arrays:
  - `rand(n)` or `rand(m,n)`
  - `randn(n)` or `randn(m,n)`
  - `randi(k,n)` or `randi(k,m,n)`

Use round brackets, `(` and `)`, to access a particular element of a vector or array.

In MATLAB, all arrays are indexed from 1.

That means, the first element of any vector, `v`, is `v(1)`.

And the first element of any matrix, `A`, is `A(1,1)`.

There is a special keyword `end` to access the final element of a vector, so that you don't have to know how many elements it has:

```
>> v = [1 2 3 4]
v =
     1     2     3     4
>> v(1)
ans =
     1
>> v(end)
ans =
     4
```

A very powerful feature of MATLAB is that you can use integer vectors to access multiple entries at once.

```
>> v = randn(1, 5)
v =
    0.3035   -0.6003    0.4900    0.7394    1.7119
>> v([3,2])
ans =
    0.4900   -0.6003
```

Vector indexing can also be used for setting values:

```
>> x = 1:10
x =
   1     2     3     4     5     6     7     8     9    10
>> x(2:2:10)=0
x =
   1     0     3     0     5     0     7     0     9     0
```

It takes a little getting used to, but one can also use logical indexing. For example, suppose we have the vector v with entries

```
v = [1, -2, 3, -4, 5, -6, -7, 8]
```

and we want to change all the negative entries to 0. Here are two ways to do that

```
for i=1:length(v)
    if (v(i) <0)
        v(i)=0
    end
end
```

Or, in a single line:

```
>> v(v<0)=0
v =
     1     0     3     0     5     0     0     8
```

- `find(v)` returns the index of all non-zero entries of v. E.g.,

```
>> v = [1, 0, 0, -2, 0, 0, 3];
>> find(v)
ans =
     1     4     7
```

- `max(v)` and `min(v)`
- `mean(v)` and `median(v)`
- And many others!

Vector indexing works for matrices too:

```
>>  A = [1,2,3; 4,5,6; 7,8,9]
A =
     1     2     3
     4     5     6
     7     8     9
>> B = 5-A
B =
     4     3     2
     1     0    -1
    -2    -3    -4
>> B([1,2],[2,3])
ans =
     3     2
     0    -1
>> B([1,2],[2,3])=8
B =
     4     8     8
     1     8     8
    -2    -3    -4
```

The colon operator is very useful when doing vector indexing:

```
>> A = randi(9, 4, 4)
A =
     5     2     1     1
     1     8     4     7
     3     5     1     8
     9     9     9     8

>> A(1:3, 1:3)
ans =
     5     2     1
     1     8     4
     3     5     1

>> A(2:2:4, 2:end)
ans =
     8     4     7
     9     9     8
```

Using the colon without limits gives you am entire row, or column:

```
>> Row1 = A(1,:)
Row1 =
     5     2     1     1
>> Col2 = A(:,2)
Col2 =
     2
     8
     5
     9
```

You can combine matrices and vectors to make larger ones, so long as the sizes make sense. E.g., for examples above, we could set

```
>> B = [eye(3), zeros(3,2); ones(2,3), rand(2,2)]
B =
    1.0000         0         0         0         0
         0    1.0000         0         0         0
         0         0    1.0000         0         0
    1.0000    1.0000    1.0000    0.9575    0.1576
    1.0000    1.0000    1.0000    0.9649    0.9706
```

The arithmetic operators `+`, `-`, `*` and `^` all work in the usual matrix way.

```
>> A = [2 2; 6 4]
A =
     2     2
     6     4
>> B = [-2 1; 3 -1]/2
B =
   -1.0000    0.5000
    1.5000   -0.5000
```

See what you get with, for example `A+2*B`, `B*A`, `A^2`, etc.

Note that `A^2` is the same as `A*A`.

Entry-wise operations are done by putting a "dot" before the operator.
Compare `A^2` with `A.^2`

There are certain matrices that are very important in particular areas, and there a MATLAB functions to build them. Examples (which we will not dwell on) include `toepliz`, `hankel`, `hadamard`, `hilbert` and `vander`.

My favourites are : `magic` and `pascal`.

```
>> magic(3)
ans =
     8     1     6
     3     5     7
     4     9     2
```

```
>> pascal(5)
ans =
     1     1     1     1     1
     1     2     3     4     5
     1     3     6    10    15
     1     4    10    20    35
     1     5    15    35    70
```

MANDEL.m

```
%% MANDEL Mandelbrot set.
% Taken from Listing 1.4 of The MATLAB Guide, 3rd Ed
% https://epubs-siam-org.nuigalway.idm.oclc.org/doi/pdf/10.1137/1.9781611974669.ch1
h = waitbar(0,'Computing...');
x = linspace(-2.1,0.6,2001);
y = linspace(-1.1,1.1,2001);
[X,Y] = meshgrid(x,y);
C = complex(X,Y);
Z_max = 1e6; it_max = 50;
Z = C;
for k = 1:it_max
   Z = Z.^2 + C;
   waitbar(k/it_max)
end
close(h)
contourf(x,y,double(abs(Z)<Z_max))
colormap([1 1 1; 1/2 1/2 1/2]) % Gray inside, white outside.
title('Mandelbrot Set','FontSize',16,'FontWeight','normal')
```