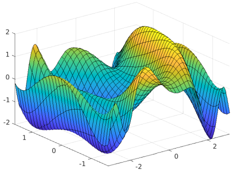
CS319: Scientific Computing (with MATLAB)

Scripts, Functions, and Matrices

Niall Madden

Week 4: **9am and 4pm**, 01 Feb 2023



Other reading:

- Chapter 5 of The MATLAB Guide: https://doi-org.nuigalway.idm.oclc.org/10.1137/1.9781611974669
- Chapters 2, 3 and 4 of Learning MATLAB: https://doi-org.nuigalway.idm.oclc.org/10.1137/1.9780898717662

This week, in CS319:

Am

1. 1: Scripts
   - Live scripts
2. 2: Anonymous functions
   - Anonymous functions
   - Funfuns
3. 3: "File functions"
   - Workspaces
   - Recursion
   - Multiple outputs
4. 4: Vectors (again)
   - Common vectors
   - Accessing elements
   - Vector indexing
5. 5: Matrices (again)
   - Common matrices
   - Special matrices
   - Accessing elements
   - Vector indexing
   - Arithmetic Operations
   - Sparse matrices
   - Other matrix functions
6. 6: Matrix division

PM.

1: Scripts

A **script** is a collection of MATLAB instructions gather into a file.

Typing the file's name (or clicking on "Run") in the editor, is the same as typing in the list of instructions.

As we learned in Lab 1, a script name must start with a letter, and can include letters, digits, and the underscore symbol.

Common mistake: using a space or a minus sign in a script's name.

also: file name ends with .m.

Some pointers:

- Include comments, with percent signs, at the start of the script. These will be shown if you type "`help` *`file_name`*" in the command window.
- These comments also appear as text if you export your script. (Other mark-up is possible; will discuss another day).
- `clear:` It is a good idea to have the keyword `clear` at the start of a script so that any old variables can be removed.
- Can also use `clear` *`variable-name`* to clear a variable, and free up some memory.

Scripts contain just code and comments.

MATLAB also features a notebook-type environment called "**Live Scripts**". These mix formatted text (include mathematics, written in LaTeX), with code.

We'll use Live scripts extensively later in the module.

2: Anonymous functions

A function is a like a formula to preform computations. Really, it is a way of converting some inputs to outputs.

MATLAB has two types of function:

- Anonymous functions, which are very simple, usually just have a single line of code.
- "File functions": usually have multiple files, and are stored in separate files.

Also called "@"-functions.

A simple function can be defined using the "@" symbol.

Syntax is : *fn = @(vars)formula*.

```
>> f = @(x)sin(1./x) % note use of entrywise operator
f =
  function_handle with value:
    @(x)sin(1./x)
>> fplot(f,[0.01, .1])
```

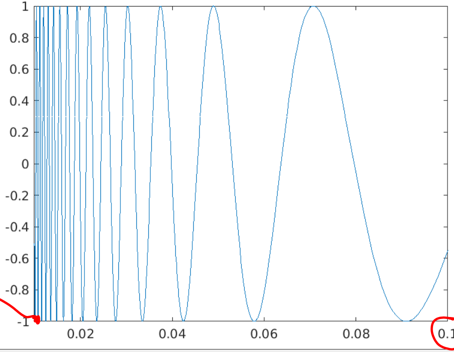
Syntax
fplot(fn_name, [domain])

Vector:
[a, b]

f maps values of x to other values.

- x is a place-holder
- x is "local".

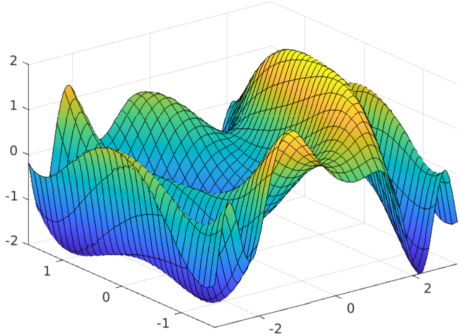
a



b.

Functions can be multidimensional:

```
>> f = @(x,y)sin(x+y.^2) - cos(y-x.^2)
f =
  function_handle with value:
    @(x,y)sin(x+y.^2)-cos(y-x.^2)
>> fsurf(f, [-pi pi -pi/2 pi/2])
```



There are various built-in functions that work directly with anonymous functions ("funfun"). For example,

```
>> fplot(f,[0.01, .1]); % plot a 1D function

>> fsurf(f, [-pi pi -pi/2 pi/2]) % plot a 2D function

>> fmesh(f, [-pi pi -pi/2 pi/2])  % plot a 2D function

>> fminbnd(g, -1,2)  % find the minimum of a function on an
      interval

>> integral ...

>> doc funfun
```

also fzero

3: "File functions"

For more complicated functions, that have multiple lines of code, we can store the code in its own file:

- The name of the function is the name of the file (excluding the "doc m").
- first word in the file is the keyword `function`
- Syntax:

```
function ReturnVal = FunctionName(param list)
```

- The return value can be a vector of any type or size, and can even be a vector of other vectors or matrices!
- The function returns whatever value this variable has at the end of the file.
- The parameter list is separated by commas.
- You can include functions in script files, which are then associated with just that script; but we'll until working with Live Scripts before using this feature.

IsComposite.m

```
function answer = IsComposite(d)
% Check if the integer argument d is prime or composite
% return "true" or "false"
% Note: (a) one could use the "isprime" function
%       (b) This is _very_ inefficient
answer = false;
for k = 2:d-1
   if ( mod(d, k) == 0)
      answer = true;
   end
end
end
```

```
>> IsComposite(25)
ans =
  logical
   1
>> IsComposite(5)
ans =
  logical
  0
```

1 is true

0 is false.

When working from the command line, or a script we have access only to variables in the **base workspace**, meaning variables that have been defined at the command line or in scripts (and not cleared).

Functions have their own "local" workspace. So function only have access to variables that are explicitly passed in the parameter list.

And even those that are passed, are "passed by value".

Any variable created in a function file is cleared when the function finishes, and so is not accessible to any other function, or the command line

Global variables

Exception: a variable can be defined as `global` to extend its scope. Global variables are unavoidable when writing apps, but otherwise are poor practice.

This is an example is taken from this week's lab. **Questions?**

Bisection.m

```
%%BISECTION
% Use a bisection algorithm to find a local max of
% a given function, f, on the interval [a,b]
function c = Bisection(ObjFn, a, b)

while ( (b-a) > 1e-6)
   c = (a+b)/2.0; % center
   l = (a+c)/2.0; % left
   r =(c+b)/2.0;  % right

   if ( (ObjFn(c) > ObjFn(l)) && (ObjFn(c) > ObjFn(r)) )
      a=l;
      b=r;
   elseif ( ObjFn(l) > ObjFn(r) )
      b=c;
   else
      a=c;
   end
end
```

Bisection (f, x, y)

f is mapped to ObjFn

x is mapped to a

y is mapped to b.

Also: all variables are passed by value (not reference).

Many problems in scientific computing can be solved by replacing the problem by a similar but simpler one, and solving that instead.

Here are a few very simplistic examples:

- Suppose we want to compute $x = a^b$, where b is a positive integer. We could first compute a^{b-1}, and then set $x = (a)(a^{b-1})$. The process can be repeated:

  $a^{b-1} = (a)(a^{b-2})$ so $x = (a)(a)\, a^{b-2}$.

- Suppose we want to compute $x = n!$, where n is a positive integer. We could first compute $(n-1)!$, and then compute $x = (n)(n-1)!$.

Both these are candidates for computation by recursion.

Power.m

```
%% POWER.m Example of a recursive function
% It uses a function defined in a script.
% Older version of MATLAB do not permit this.

a = 3.1;
b = 5;
c = RecursivePower(a,b);
fprintf("%f to the power of %i = %f\n", ...
    a, b, c);

%% Compute y = x^p
function y = RecursivePower(x, p)
if (p==0)
    y = 1;
else
    y = x*RecursivePower(x, p-1);
end
end
```

Finished here at 10am.

A less trivial example: a recursive decimal-to-binary converter. *Can you work out how it works?*

mydecimal2binary.m

```
function bin = mydecimal2binary(dec)
%% DEC2BINARY.m a decimal to binary converter
if (dec <= 1)
   bin = dec;
else
   bin = 10*mydecimal2binary( floor(dec/2) ) + mod(dec, 2);
end
```

Most common functions compute a single output. However, often functions return more than one value: see Lab 2 for an example of this.

The syntax is:

```
[val1, val2, val2] = MultiFunction(x)
```

And in the function file:

```
  [r1, r2, r3] = function MultiFunction(x)
  ...
```

Val1 assigned value of r_1

val2 assigned value of r_2

Val3 , " " " r_3

Note: val2 & val3 are optional.

It is legal to call the function without using all the return values.

An example of this is the MATLAB max function. Example:

```
>> x = 0:0.25:1
x =
          0    0.2500    0.5000    0.7500    1.0000
>> v = x.*(1-x)
v =
          0    0.1875    0.2500    0.1875         0
>> max(v)
ans =
     0.2500
>> [max_x, max_i] = max(v)
max_x =
     0.2500
max_i =
     3
```

max returns maximum value of a vector, and its index.

Another example: the `sort` function

List(key(i)) = S(i)

```
>> List = randi(10,1,6)
List =
     7     4    10     1     5     4
>> S = sort(List)
S =
     1     4     4     5     7    10
>> [S,key]=sort(List)
S =
     1     4     4     5     7    10
key =
     4     2     6     5     1     3
```

Note that `List(key)` gives the sorted list, S.

. . . .

MATLAB also allows for functions that have variable numbers of inputs and outputs, using `nargin`, `nargout`, `varargin` and `vargout`. We'll discuss these later if we need them.

3: “File functions” — Example

This example (taken from Chap 3 of Learning MATLAB) show how/way one would have multiple return values.

QuadRoots.m

```
%% QuadRoots: Compute the roots of a*x^2 + b^x + c
% Taken from "Learning MATLAB", Chapter 3
function [x1, x2] = QuadRoots(a,b,c)
d = sqrt(b^2 - 4*a*c);
x1 = (-b + d)/(2*a);
x2 = (-b - d)/(2*a);
```

```
>> A=1; B=5; C=6;
>> x1=QuadRoots(A,B,C)
x1 =
    -2
>> [x1,x2]=QuadRoots(A,B,C)
x1 =
    -2
x2 =
    -3
```

If we only want the second return value:

[~, r2] = QuadRoots(A,B,C)

then first return value is ignored.

4: Vectors (again)

A vectors (in MATLAB) is either

- $1 \times n$ array, for a row vectors
- $n \times 1$ array, for a column vectors

The simplest way to define a vector 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

```
>> b = [5 5 5]
b =
     5     5     5
```

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

There are functions to construct some standard vectors.

- `Z = zeros(1,n)` is the $1 \times n$ zero vector.
- `ones(m,1)` returns the $m \times 1$ vector of all ones.
- `linspace(x1, x2, n)` returns the linear spaced (row) vector with n entries, and successive entries differ by $(x2 - x1)/n$.
- The colon operator, which we used last week for `for`-loops, defines row vectors:

`x=a:b` $\rightarrow$ $x = [a, a+1, a+2, \ldots, b]$

`x=a:h:b` $x = [a, a+h, a+2h, \ldots, b]$

```
>> v = 1:4
v =
     1     2     3     4

>> w = 0:2:10
w =
     0     2     4     6     8    10

>> x = 100:-10:0
x =
   100    90    80    70    60    50    40    30    20
        10     0

>> y = 1:-1:10
y =
  1x0 empty double row vector
```

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

In MATLAB, all vectors are indexed from 1.

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

Like v(-1) in Python.

To check the number of entries in a vector use : `length(v)` or `size(v)`.

If v is a vector, and x is a vector with integer entries between 1 and `length(v)`, then `w=v(x)` is a vector:

- `length(w)` is equal to `length(x)`
- `w(i)` has value of `v(x(i))`.

```
>> v = [3.14, 2.1, 3.333, 0.4, 0.5];
>> x = [1, 5, 2, 2];
>> w = v(x)
w =
    3.14    0.5    2.1    2.1
```

5: Matrices (again)

(We covered the material in these slides last week. They are just here for completeness and so I can highlight certain ideas.)

A matrix is the default data type in MATLAB.

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
```

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

There are functions to construct some standard matrices.

- `I = eye(N)` makes the $N \times N$ identity matrix
- `Z = zeros(m,n)` is the $m \times n$ zero matrix. `zeros(n)` is the same as `zeros(n,n)`.
- `ones(m,n)` returns the $m \times n$ matrix, all of whose entries are 1.
- Random arrays:
  - `rand(n)` or `rand(m,n)`
  - `randn(n)` or `randn(m,n)`
  - `randi(k,n)` or `randi(k,m,n)`

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
```

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
```

$8 + 1 + 6 = 5$

$\parallel$ 15

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 2 & 1 & 0 \\ 1 & 3 & 3 & 1 \end{pmatrix}$$

"Rotate"

```
>> 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
```

$$\begin{array}{ccccccccc} & & & & 1 & & & & \\ & & & 1 & & 1 & & & \\ & & 1 & & 2 & & 1 & & \\ & 1 & & 3 & & 3 & & 1 & \\ 1 & & 4 & & 6 & & 4 & & 1 \end{array}$$

Again, use round brackets, `(` and `)`, to access a particular element of a matrix: `A(i,j)` returns the entry in row *i*, column *j* of `A`.

As noted, all arrays are indexed from 1, so the first element of the matrix, `A`, is `A(1,1)`.

```
>> M = magic(4)
M =
    16     2     3    13
     5    11    10     8
     9     7     6    12
     4    14    15     1
>> M(1,end)
ans =
    13
>> M(end,2)
ans =
    14
>> M(end,end)
ans =
    1
```

Vector indexing works for returning and setting multiple entries of a matrix at once.

```
>>  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
```

sets $B(i,j) = 5 - A(i,j)$.

```
>> A = pascal(4)
A =
      1     1     1     1
      1     2     3     4
      1     3     6    10
      1     4    10    20
>> A(1:3, 1:2)
ans =
      1     1
      1     2
      1     3
>> A(2:3, :)=0
A =
      1     1     1     1
      0     0     0     0
      0     0     0     0
      1     4    10    20
```

Note: A(1:3, 2:) is not allowed.

Use A(1:3,2:End)

That last example showed that the colon operator without limits gives you an entire row, or column:

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

If `A` is a matrix, then `A(:)` returns the "vector" version of the matrix:

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

To reverse the process, you can use the `reshape()` function. Look it up, if interested.

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

In many practical applications, one works with matrices whose entries are mostly zero. There are special ways to store these so-called "sparse matrices".

But this is such a major topic, we will spend an entire class on it later in the semester.

Finished here 5pm

- `inv(A)`

- `det(A)`

- `A'` is the transpose of A

- `eig(A)` estimate the eigenvalues and eigenvectors of A.

And there are lots of other functions that you may have met in a linear algebra module, but we wait until we need them.

6: Matrix division

For scalars (i.e, 1×1 matrices), "division" is well understood: we know what `a/b` means $\frac{a}{b} = ab^{-1}$. This is called "right division" in MATLAB.

MATLAB also has "left division": `a\b` means $\frac{b}{a} = a^{-1}b$.

The reason for this, is that, if `A` is a matrix, and `b` and `x` are vectors so that $Ax = b$, then, of course $x = A^{-1}b$.

Solving $Ax = b$

In MATLAB, if you are given a matrix A and vector b, then we usually solve $Ax = b$ with "backslash":

```
>> x = A\b
```

6: Matrix division

It is important to note that the "backslash" operator is highly optimised. And it does not compute the inverse of a matrix. More likely, it uses Gaussian elimination, or some variant (depends on the matrix).

MATLAB can invert a matrix, using the `inv(A)` function. But if you just wish to solve a linear system, backslash is is faster, and uses much less memory.

6: Matrix division

Eg01_soler_timer.m

```
for n=2.^(2:11)
    A = randn(n);
    b = ones(n,1);
    % Test matrix left divide
    mld_start = tic;
    x = A\b;
    mld_time = toc(mld_start);
    % Test inv()
    inv_start = tic;
    B = inv(A);
    x = B*b;
    inv_time = toc(mld_start);

    fprintf('n=%4d. MLD time=%6.3fs, inv time=%6.4fs (Speed
        up = %5.2f)\n', ...
      n, mld_time, inv_time, inv_time/mld_time);
end
```

Try this. I get a speed-up of a factor of about 2.5.