HM811: Computational Tools for Research

Introduction to MATLAB

Semester 2, 2013-2014

Outline

- Command Line and Workspace
- Numerical Variables
- Characters and Logical Variables
- Operations and Functions

Command line

- We can use MATLAB in two basic ways: via the command line; or as a programming language via scripts and functions.
- MATLAB is particularly effective for manipulations involving vectors and matrices and arrays of all sorts.
- The help function is particularly effective and contains information on all of the basic functions available in MATLAB.
- There are also a variety of supports and tutorials available at the mathworks site.

`www.mathworks.co.uk/academia/student_center/tutorials/launchpad.html`

MATLAB's command line

- To use the command line, simply start typing at the prompt >>.
- In MATLAB, we create variables by using them! You do not need to declare them. MATLAB automatically allocates memory.

  ```
  a = 0.3
  ```

  creates a variable *a* in the workspace and assigns it the value 0.3.
- On the other hand

  ```
  myString = 'Hello there'
  ```

  creates a variable `myString` and assigns it the string (char array) `Hello there`.

Variables

- Variable names must start with a letter and are case sensitive

  `myVar` is different to `myvar`

- In MATLAB, certain identifiers are reserved or used for a specific purpose:

  `ans; Inf; pi; NaN; i; j`

  although it is sometimes ok to use i, j as variable names where there is no risk of confusion.

Variables

- `ans` is reserved for the outcome of a calculation when no assignment is made.
- `Inf` represents inf or any number larger than the largest floating point number that MATLAB can represent. You can find out what this is with the `realmax` function.
- `NaN` stands for *not a number* and arises in calculation such as $0/0$. It can also represent missing data.
- i, j are both used for the imaginary $\sqrt{-1}$. You can also assign them a value however.

Variables

- We can use previously defined (and initialised) variables to define and initialise new ones.

```
x = 0.3;
y = x^2 + 2*x + 3;
```

- We use a semicolon at the end to prevent the value of the expression being displayed as `ans = 3.69`.

Workspace

- As well as the command line, the main window also contains the Current Folder and the Workspace.
- Current Folder: You can load data from the files in here or run an scripts/functions stored here.
- Workspace: The active variables you have created/assigned or loaded from files.
- There are some useful commands for working with the command window.

Workspace

- `clear` clears all variables from the workspace. `clear all` clears variables, functions and links.
- `clear v1 v2 v3` clears the variables `v1, v2, v3`.
- `whos` displays information on all variables currently in the workspace.

Workspace

- You can save variables from the Workspace to a file and load variables back in using the `save` and `load` commands.
- `save 'myfile' myvar1 myvar2` saves the variables `myvar1, myvar2` in a file named myfile.mat
- `load 'myfile'` reloads the variables in myfile.mat into the workspace.

Numerical Variables

- Real numbers are stored in standard double format in MATLAB.
- The largest number that can be represented this way is stored in `realmax`; the smallest in `realmin`
- You can also represent complex numbers in MATLAB.
- `complex(1, 2)`, `1+2i`, `1+2j` all represent the same number $1 + 2i$.
- `real(z)`, `imag(z)` return the real and imaginary parts of the number z. The phase of z is given by `angle(z)`.

Strings and Logical Variables

- `a = 'hello'` creates a string or character array with 5 entries.
- MATLAB is essentially array-based - scalars or characters are arrays with a single element.
- logical variables take the values `true` and `false`. These are represented as 1 and 0 but use less space than the double representation.

Logical Operations

There are various operations we can carry out on logical variables.

- `&` (AND): `L1 & L2` true if both `L1`, `L2` are true.
- `|` (OR): `L1 | L2` true if either of `L1`, `L2` are true.
- `~` (NOT): `~L1` true if `L1` if false.

Comparison Operators

- `a==b` (scalars): evaluates to true when `a` and `b` are equal.
- `a<b` (scalars)
- `a<=b` (scalars)
- When the variables are scalars, these output a single scalar logical variable: care should be taken when dealing with arrays/matrices.
- Logical functions such as `isreal`, `isfinite`, `isnan`, `isinf` can be used to test whether variables contain real values, finite values etc.

Arithmetic and Functions on Scalars

`+, -, *, /`	addition, subtraction, multiplication, division.
`^`	exponentiation (raising to a power)
`abs, real, imag`	absolute value, real and imaginary parts
`log, log2, log10`	natural, base 2, base 10 logarithms
`exp`	usual exponential function
`sqrt`	square root function
`sin, cos`	sin and cosine functions
`ceil, floor, round`	ceiling, flooring and round functions.

You can find information on elementary functions and special functions in MATLAB by typing `help elfun, help specfun`.

Outline

- Vectors, Matrices and Operations on them
- Strings and Character Arrays
- Multidimensional Arrays
- Cell Arrays
- Structures

Vectors and Matrices

- MATLAB is particularly well suited to calculations involving arrays of numbers arranged as vectors or matrices.
- Row Vectors: List components between two square brackets `[, ]`; separate components by white-space or commas.

```
r = [2 -1 2]
```

- Column Vectors: Separate components by semi-colons. A semi-colon indicates a new line.

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

- Indices in MATLAB start at 1.
- To access a particular component in a vector:

```
t = r(2);
```

`t` will take the value -1.

Vectors and Matrices

- Matrices are defined similarly. The elements within each row are separated by spaces or commas, rows are separated by semi-colons.

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

- The r, c element of a matrix A is accessed as $A(r, c)$. So `A(2, 3)` above will take the value 2.
- $A(r, :)$ is the rth row of A.
- $A(:, c)$ is the cth column of A.

Matrices

- $A(:)$ represents A as a column vector, going through A column by column. It also provides a way of making sure that a vector is a column vector.
- `v = v(:)` ensures that `v` is a column vector.
- $A([1,2],[2,3])$ picks out the submatrix of A corresponding to rows 1 and 2 and columns 2 and 3.
- Similarly `A([1:3], [2:5])` selects the submatrix of A corresponding to rows 1-3 and columns 2-5.

Dynamic Allocation

- If you assign a value to an index outside the current range of a vector or matrix variable MATLAB resizes appropriately.

For example, try typing:

```
v = [1 2 1];
v(6) = 3
```

You can also remove a row or column from a matrix by typing `A(2, :) = []` or `A(:, 2) = []`

Operations on Vectors and Matrices

- `*, +` is the usual vector-matrix multiplication and addition.
- `v', A'` calculates the conjugate transpose of `v, A`.
- For a vector `length(v)` returns the number of components or the dimension of the vector.
- `max(v)` returns the maximal element in `v`. If we write `[m, i] = max(v)`, then `i` contains the index of the maximal element. `min` works similarly.
- `sum(v)` sums the elements of `v`

Operations on Vectors and Matrices

- For a matrix A, `size(A)` returns the ordered pair (R, C) where R is the number of rows in A and C is the number of columns.
- `size(A, 1)` returns the number of rows in A. `size(A, 2)` returns the number of columns in A.
- If A is $m \times n$ then `max(A)` returns a row vector containing the maximal element in each column.
- `sum(A)` returns a row vector containing the sum of each column.
- `A^p` raises A to the power of p - usual matrix multiplication.

Operations on Vectors and Matrices

- To multiply two matrices elementwise, we place a dot before *. So `A.*B` represents the elementwise product of A and B.
- Similarly $A./B$ is the elementwise quotient of A and B.
- The functions `log, exp` operate elementwise on a matrix or vector.

```
>> exp([0, 1])

ans =

    1.0000    2.7183
```

- The usual matrix exponential $e^A = I + A + \frac{A^2}{2!} + \cdots$ is given by `expm(A)`.

Initialising Vectors and Matrices

It is possible to set entire vectors or matrices to initial values in a variety of efficient ways.

- `v = 1:10` sets `v = [1 2 3 4 5 6 7 8 9 10]`.
- The general syntax for initialising a vector as above is `v = lower:step:upper`. The default step size is 1.
- `A = zeros(n)` sets A to be the $n \times n$ zero matrix. For rectangular matrices (and vectors) use `A = zeros(m, n)`.
- `A = ones(n)` sets A to be the $n \times n$ matrix of all ones. Same comment as above for rectangular matrices.
- `A = eye(n)` sets A to be the $n \times n$ identity matrix.

Initialising Vectors and Matrices

- `rand(m, n)` generates an $m \times n$ matrix with entries drawn from the uniform distribution on $[0, 1]$.
- `rand(n)` generates an $n \times n$ matrix.
- `randn()` works similarly but the entries are drawn from the standard normal distribution.
- `randi( K, [m, n])` generates a matrix with integer entries (again from the uniform distribution) in the range `1:K`.
- Matrices can also be specified in block form in MATLAB. The block dimensions must agree however.

Operations on Vectors and Matrices

- The function `find()` can be very useful.
- For a vector `v`, `find(v)` returns the indices of the non-zero entries in `v`.
- `find` also works on logical expressions. For instance

  `find(v>0)`
- Recall that given a matrix A, the syntax `A(:)` vectorises A by stacking the columns on top of each other.
- If we apply `find` to a matrix, it works on the vectorised form.

Logical Indexing

- It is possible to select the elements of an array satisfying a particular condition by logical indexing
- For example: `v = [1 -1 0 1 2 -3]`.
- `v((v~=0))`, `v((v>0))`, `v((v<0))` return the non-zero, positive and negative entries in `v` respectively.
- This can be useful for preprocessing data - more on this later.

Comparing Arrays and Matrices

- When comparing vectors and matrices, we need to be careful.
- The statement `A == B` returns a matrix of logical entries showing where `A` and `B` are equal. The same is true for the other comparison operators.
- `isequal(A, B)` tests whether they are equal as matrices.
- The functions `all` and `any` are very useful here.

Multi-dimensional Arrays

- It is also possible to define arrays with more than 2-indices - corresponding to tensors.
- These can be useful for storing data from sensor networks and other similar sources.
- The functions `ones`, `zeros`, `rand`, `randn` all work for multi-dimensional arrays.
- `sum(A, d)` sums over the values of the dth index.

Cell Arrays

- Cell Arrays allow us to store arrays of different sizes and types in one overall array.
- To create a cell array, specify the elements between curly braces `A = {[1, 2], 'hello', rand(5)}`.
- To access the elements of a cell array, you must use curly braces. `A{2}`.
- Cell arrays provide a useful way of storing strings of different lengths in a single array.

Cell Arrays

- `SArray = ['Hello' 'World' 'How' 'Are' 'You?']` creates a concatenated string (without spaces);
- If we try `SArray = ['Hi'; 'There']` we will get an error as each "row" does not contain the same number of entries.
- Two alternatives for this: character arrays `S = char('Hi', 'There')` or use a cell array.
- Also note the difference in indexing between character arrays and cell arrays.

Structures

- MATLAB also supports structures: a data array with multiple dimensions named with textual identifiers.
- If we wished to store information on books for instance we could use something like

```
>> Str.title = 'Calculus';
>> Str.author = 'A. Maths';
>> Str.Year = '1667';
```

- We can also create arrays of structures by adding new entries `S(2)` etc.

Outline

- Summary Statistics
- Preprocessing
- Simple Plotting

Summary Statistics

MATLAB allows us to calculate all of the main summary statistics for data sets.

- `mean(v)`: the arithmetic mean of the entries of `v`.
- `std(v)`, `var(v)`: the sample standard deviation and variance (normalising by $n-1$ as default).
- `median(v)`, `mode(v)` compute the median entry and the mode (most frequent).
- `hist(v)` plots a histogram of the data in `v`.
- For a list of the basic data functions available in MATLAB type `help datafun`.

Simple Preprocessing

- The logical indexing in MATLAB can be helpful to *clean up* a data set.
- A non-response or absence of data can be recorded as `NaN`.
- To remove these entries from a data vector `v`, we can type `w = v(~isnan(v))`.

Simple Preprocessing

- It is also possible to remove outliers. These are often defined as data points more than 3 standard deviations from the mean.
- To identify these and remove them from a data vector `w` we can proceed as below:

```
>> mu = mean(w);
>> sig = std(w);
>> outliers = abs(w - mu) > 3*sig;
>> wclean = w(~outliers);
```

Simple Plotting

- We can plot several data sets together on a histogram. If each set is a column in a matrix `D`, simply pass the matrix to `hist(D)`.
- We can also add a legend by typing `legend('Col 1', 'Col 2', 'Col 3');`.
- When studying the relationship between two variables or data sets, it can be useful to sketch a scatter plot.
  `scatter(D(:, 1), D(:, 2), 'filled');`
- `cov(d1, d2)` and `corrcoef(d1, d2)` calculate the covariance matrix and correlation coefficient between the two data vectors.

Outline

- Simple 2-d plots
- Multiple functions in one plot
- Subplots
- 3-D Plots

Plotting

- When we have two vectors `x, y` of the same length, we can plot one against the other.
- At its simplest, all that is needed is

```
plot(x, y)
```

- We can add an x and y label using the commands

```
xlabel('This is an x-label');
ylabel('This is a y-label');
```

- To add a grid or title use the commands

```
grid;
title('My Title')
```

Plotting

For example to plot the *sin* function from 0 to 2π, we would type:

```
>> x = [0:pi/50:2*pi];
>> y = sin(x);
>> plot(x, y);
>> grid
>> xlabel('x');
>> ylabel('sin(x)');
>> title('Plot of Sine Function');
```

Plotting

- It is also possible to draw multiple functions on the one plot.
- For example, if we want to plot $sin(x), cos(x)$ and $sin(2x - \pi)$

```
>> x = -pi:pi/100:pi;
>> y1 = sin(x);
>> y2 = cos(x);
>> y3 = sin(2*x - pi);
>> plot(x, y1, x, y2, x, y3);
```

Plotting

- We can customise the style of our plot by specifying a colour, linestyle and marker
- For example

  ```
  plot(x, y1, 'r--+');
  ```

  specifies to use a red dashed line and place a + at each data point.
- If you don't specify a marker, none is added. Similarly if you specify a marker but no line style, only markers are drawn.

Subplots

- We can put several plots in the one figure using the subplot command.
- We create a matrix of smaller plots, specifying the indices for each.
- The basic syntax is `subplot(m,n,p)`: m and n specify the number of rows and columns in the plot and p specifies the active plot.
- We count along row by row.
- The subplot command can also be used to add title, labels and text to each subplot.

3-D Plots

- If we wish to draw a 3-d plot of a function $f(x, y)$, the 2 simplest functions are `mesh` and `surf`.
- We use `meshgrid` to define the grid of (x, y) points over which to draw the function.
- We can use `mesh` or `surf` to draw the plot.

```
>> [X, Y] = meshgrid(-pi:pi/50:pi);
>> Z = sin(X.^2+Y.^2);
>> mesh(X, Y, Z);
```

3-D Plots

- The call to `meshgrid` generates the 2-d grid at which the function is to be evaluated for the plot.
- Use of the `.` for elementwise operations is important here.
- `mesh` simply colours the lines between the points in the plot. If we use `surf`, we will colour in the connecting faces also.

3-D Plots

- To plot 3-d data without requiring it to be a surface, we use the `plot3` function.
- To plot a helix, we would type:

```
>> t = 0:pi/10:20*pi;
>> plot3(sin(t), cos(t), t);
```

- A spiral:

```
>> plot3(t.*sin(t), t.*cos(t), t);
```

Outline

- Functions in MATLAB
- Loops
- Conditional statements

Functions

- You can define your own functions in MATLAB - these are saved as m-files.
- Start the editor from the command window by typing `edit`.
- The syntax for declaring the function is

  ```
  function [out1, out2, ... ] = myFunc(in1, in2, ...)
  ```

- As always, comments are helpful. In MATLAB, anything coming after % is treated as a comment.
- Variables are handled in the same way as on the command line.

Functions

```
function n = LNorm(v)

%Returns the Euclidean Norm of the column vector v
%This is most easily given as the square
%root of the inner product of v with itself

n = sqrt(v'*v);

end
```

Functions

- As with other programming languages, MATLAB supports for and while loops and conditional structures.
- The basic syntax of the 'for' loop is

```
for i = lower:step:upper
Statements;
end
```

- The default step size is 1

Functions

```
function L = LNorm(v)

%Returns the Euclidean Norm of the column vector v
%We can also calculate this directly using the
%components of v and a for loop

n = length(v);
SumOfSquares=0;
for i = 1:n
    SumOfSquares = SumOfSquares + v(i)^2;
end
L = sqrt(SumOfSquares);
```

Functions

The basic syntax of the `while` loop is

```
while CONDITION
statements
end
```

```
function r = DivAlg(p, q)

%apply division algorithm to integers p and q to find remainder when we
%divide p by q (assume p > q)

r = p;

while r >= q
    r = r - q;
end
```

Functions

- Our first function to compute the Euclidean norm of v assumed that the vector was a column vector.
- We could verify that this is the case using an 'if' statement.
- The basic comparison operators in MATLAB are

```
==  for equality
~=  not equals
<  less than
<=, >, >= and so on...
```

If Statement Example

```
function L = LNorm(v)

%Returns the Euclidean Norm of the column vector v
%Let's check the size of v first
if size(v, 2) == 1
    L = sqrt(v'*v);
elseif size(v, 1) == 1
    L = sqrt(v*v');
else
    disp('This is not a vector');
    L = NaN;
end
```

The `switch` statement allows us to choose between a larger number of alternative options.

```
function mySt = IrishDay(m_day)

%translate the string m_day into the Irish for the same day

switch lower(m_day)
    case 'monday'
        mySt = 'Luan';
    case 'tuesday'
        mySt = 'Mirt';
    case 'wednesday'
        mySt = 'Cadaoin';
    case 'thursday'
        mySt = 'Dardaoin';
    case 'friday'
        mySt = 'Aoine';
    case 'saturday'
        mySt = 'Satharn';
    case 'sunday'
        mySt = 'Domhnach';
end
```

Outline

- Solving Linear Equations
- Eigenvalues and Eigenvectors
- Stable Matrices
- Decompositions

Linear Algebra and MATLAB

- MATLAB makes it very easy to solve systems of linear equations

$$Ax = b$$

where $A \in \mathbb{R}^{m \times n}$, $b \in \mathbb{R}^m$ are given.

- MATLAB solves such systems using

```
x = A\b
```

- Whether an exact solution exists and is unique depends on the data given.
- Systems can be over-determined or undetermined or have a unique solution.
- The solution MATLAB gives depends on which situation pertains.

Unique Solution

- To check that there exists a solution, verify that `rank([A, b]) = rank(A)`.
- If in addition `rank(A) == n`, the solution is unique.
- For example, suppose

$$A = \begin{pmatrix} 1 & 2 \\ -1 & 1 \end{pmatrix}, b = \begin{pmatrix} 1 \\ 2 \end{pmatrix}.$$

```
>> A = [1 2;-1 1];
>> b = [1;2];
>> rank([A b]) - rank(A)

ans =

     0
```

Unique Solution

So this system has a unique solution.

```
>> x = A\b

x =

     -1
      1
```

We can also use the inverse as A is square in this case.

```
>> inv(A)*b

ans =

   -1.0000
    1.0000
```

Unique Solution

Of course, we should first check that the matrix is non-singular by calculating its determinant.

```
>> det(A)

ans =

     3
```

As another example/exercise, consider

$$A = \begin{pmatrix} 1 & 2 \\ -1 & 1 \\ -1 & 4 \end{pmatrix}, b = \begin{pmatrix} 4 \\ -1 \\ 2 \end{pmatrix}.$$

Undetermined Systems

- Suppose `rank([A, b]) = rank(A)` and `n > rank(A)`.
- The system is said to be undetermined. There will be multiple solutions.
- Consider

$$A = \begin{pmatrix} 1 & -1 & 2 \\ 2 & 0 & -1 \end{pmatrix}, b = \begin{pmatrix} 1 \\ 1 \end{pmatrix}.$$

```
>> rank([A b]) - rank(A)

ans =

     0
```

Undetermined Systems

There will exist a solution to this system. However as the number of unknowns, 3, is greater than the rank of A, this solution will not be unique.

```
>> A\b

ans =

    0.6000
         0
    0.2000
```

yields one solution. It is also possible to use

```
pinv(A)*b
```

to find an alternative solution (the one with minimal l_2 norm in fact). The function `pinv` gives the pseudo-inverse of A.

Overdetermined Systems

- If the number of unknowns is less than the rank of `[A, b]`, then the system is over-determined and there may not exist a solution (more independent equations than unknowns).
- In this case, MATLAB calculates a least squares solution minimising $\|Ax - b\|_2$ and such that $\|x\|_2$ is minimal also.
- As an example try the system with

$$A = \begin{pmatrix} 2 & -1 \\ 1 & 2 \\ 1 & -1 \end{pmatrix}, b = \begin{pmatrix} 1 \\ -1 \\ 1 \end{pmatrix}.$$

Overdetermined Systems

```
>> A\b

ans =

    0.2286
   -0.6286
```

You should check that the command

```
>> pinv(A) *b
```

yields the same solution.

Eigenvectors and Eigenvalues

- The `eig` function calculates eigenvalues and eigenvectors for square matrices.
- `eig(A)` returns a vector containing the eigenvalues of A.
- `[V, D] = eig(A)` computes eigenvalues and eigenvectors. D is a diagonal matrix containing the eigenvalues. The corresponding columns of V are the eigenvectors.
- As an example calculate the eigenvectors and eigenvalues of

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

Positive Definite Matrices

- A matrix is positive definite if it is Hermitian $A = A^*$ and all of its eigenvalues are positive.
- Let's write a simple function to test if a matrix is positive definite in MATLAB.

```
function t = IsPD(A)
%Checks if A is positive definite
%   returns 1 if A is PD and 0 otherwise

if (A==A') & min(eig(A)) > 0
        t = 1;
else
        t = 0;
end
```

Hurwitz or Stable Matrices

- Hurwitz matrices are important in the stability theory of Linear Systems.
- A square matrix A is Hurwitz if all of its eigenvalues have negative real part.
- Any real Hurwitz matrix will have negative trace, while the sign of its determinant will depend on its size.
- As an exercise, let's write a function to test if a matrix is Hurwitz.

Hurwitz Matrices

```
function t = isHurwitz(A)

if max(real(eig(A))) < 0
    t = 1;
else
    t = 0;
end

end
```

Hurwitz Matrices

Some Exercises

- Write a simple function that can 'randomly' generate a positive definite matrix of size n.
- Write a simple function that can 'randomly' generate a Hurwitz matrix of size n.
- Is the sum of two Hurwitz matrices necessarily Hurwitz? Test this numerically.

Schur Matrices

- Hurwitz matrices are important in the analysis of continuous time systems $\dot{x} = Ax$.
- For discrete time systems $x(k+1) = Ax(k)$, stability is equivalent to the matrix A being Schur.
- This means that all eigenvalue of A have modulus less than 1.
- Write two functions to: (i) test if a matrix is Schur; (ii) generate a Schur matrix of given size.

Schur Matrices and Stability

- Now generate an example of a Schur matrix A of size 2.
- Select an initial vector $x(0)$.
- Compute $x(1), x(2), x(3)$ and so on.
- Note the convergence properties of the iterates.

Singular Value Decomposition

- The SVD is widely used in areas including Control Theory and Information Retrieval.
- MATLAB also has built in functions for the LU, QR, and Schur decompositions.
- Given $A \in \mathbb{R}^{m \times n}$, there are unitary matrices $U \in \mathbb{R}^{m \times m}$, $V \in \mathbb{R}^{n \times n}$ and a diagonal matrix $S \in \mathbb{R}^{m \times n}$ (meaning $s_{ij} = 0$ for $i \neq j$) such that
$$A = USV^*.$$
- To compute the svd in MATLAB, use the command
`[U, S, V] = svd(A);`

Singular Value Decomposition and Pseudoinverse

- Another way of computing the pseudoinverse of A is to calculate

$$VS^{+}U^{*}$$

  where S^{+} is the 'diagonal' matrix in $\mathbb{R}^{n\times m}$ whose diagonal entries are the reciprocals of the non-zero diagonal entries in S.
- Verify this for some examples in MATLAB.

SVD and Low Rank Approximation

- Another use for the SVD is to compute low rank approximations of a matrix. This has applications in the design of information retrieval systems.
- Given $A \in \mathbb{R}^{m \times n}$, the rank of A is the number of non-zero singular values in the diagonal matrix S.
- The matrix US_kV^* where S_k is a 'diagonal' matrix containing only the largest k singular values of A is the closest rank k matrix to A, where distance is measured in the l_2 norm.

SVD and Low Rank Approximation

Let's write a simple function to compute a rank k approximation to a given matrix.

```
function [Ak] = LowRankApprox(A, k)

%Uses the SVD to compute an optimal rank k approximation to A

[m, n] = size(A);

if k>rank(A)
    disp('Rank exceeds rank of A');
    Ak=[];
else
    [U, S, V] = svd(A);
    Sk = S;
    Sk([k+1:m], [k+1:n]) = 0;
    Ak = U*Sk*V';
end

end
```

Outline

- Polynomials and fitting
- Finding zeros
- Optimisation
- Ordinary Differential Equations

A Little Calculus

- MATLAB is also very useful for polynomial fitting, optimisation and simulating ordinary differential equations.
- Let's start by discussing how MATLAB handles polynomials.
- The polynomial

$$p(x) = p_0 + p_1 x + \cdots + p_n x^n$$

is represented by the vector of coefficients

```
p = [p0 p1 ...  pn]
```

- To evaluate it at a particular value of x, type

```
polyval(p, x).
```

Polynomials

Consider the polynomial $p(x) = 1 + 2x + x^2$. Represent it in MATLAB and evaluate it at $x = 1$.

```
>> p = [1 2 1]

p =

     1     2     1

>> polyval(p, 1)

ans =

     4
>> p = [1 0 1 2];
>> polyval(p, 2)

ans =

    12
```

Write down the second polynomial above?

Polynomials

- We can also find the roots of a polynomial using the function `roots(p)`.
- The argument `p` is the vector of coefficients.
- To retrieve the coefficients from the roots use `poly(r)`.

```
>> r = roots(p)

r =

    -1
    -1

>> coeff = poly(r)

coeff =

     1     2     1
```

You should experiment with some more interesting polynomials than $(1 + x)^2$.

Polynomial Fitting

- Given data points $(x_1, y_1), \ldots, (x_k, y_k)$, find a polynomial that fits.
- One approach is to find a polynomial $p(\cdot)$ that minimises the l_2 distance $\sum_{i=1}^{k} |y_i - p(x_i)|^2$.
- In MATLAB, this is done with the function `polyfit(x, y, n)`. The parameter n specifies the degree of the polynomial.

```
>> x = [1 2 3]

x =

     1     2     3

>> y = [1.5 2.1 1.8]

y =

    1.5000    2.1000    1.8000

>> p = polyfit(x, y, 2)
```

Simple Optimisation and Equation Solving

- MATLAB also supports simple optimisation. There is also an optimisation toolbox with far more sophisticated functionality.
- We describe three simple functions, `fzero`, `fminbnd`, `fminsearch`.
- `fzero(@mFun, a)` or `fzero('mFun', a)` searches for a zero of the function `mFun` starting at the point `a`.
- We must define `mFun` in a separate file (well not really but for now bear with me).

Finding Zeros

```
function t = mFun(x)

t = sin(x) + cos(x);

end
```

Save this and then to find a zero near to 2.4, type:

```
>> z = fzero('mFun', 2.4)

z =

    2.3562

>> z = fzero(@mFun, 2.4)
```

will give the same result.

Minimising Functions

The function `fminbnd('mFun', a, b)` finds the minimum of the function `mFun` in the interval $[a, b]$.

```
>> m = fminbnd(@mFun, 1, 2)

m =

    2
```

The minimum value of mFun in the given range occurs at 2. The minimum value is 0.4932.

`fminsearch('mFun', a)` finds a local minimum of `mFun` starting at the point a.

```
>> m = fminsearch(@mFun, 1.2)

m =

    3.9270
```

Simple Functions

- It is not strictly necessary to define the function in a separate file.
- We can specify the argument and the function to evaluate within the call to `fzero`, `fminbnd`, `fminsearch`.

For example:

```
>> m = fminbnd(@(x)(sin(x)*cos(x)), 0, 2*pi)

m =

    2.3562
```

Ordinary Differential Equations

- MATLAB has a number of functions that can be used to solve ODEs of the form
$$\dot{x}(t) = f(x(t)).$$
- The most popular is `ode45`.
- The basic syntax is

```
[t, y] = ode45('mRHS', [a, b], y0)
```

- `mRHS` is the function defining the right hand side of the ODE.
- `[a, b]` is the range over which the equation is to be solved.
- `y0` is the vector (column) of initial conditions.
- `t` contains the time points while `y` contains the corresponding solution values.

ODEs

- Solve the linear system $\dot{x} = Ax$ where

$$A = \begin{pmatrix} -5 & 2 \\ 1 & -3 \end{pmatrix}.$$

- First define the right hand side as a function. Be careful that the return value is a column vector.

```
function dydt = Lin(t, y)

dydt = zeros(2, 1);

dydt(1) = -5*y(1)  + 2*y(2);
dydt(2) = y(1) - 3*y(2);

end
```

ODEs

We then call ode45 to solve the equation and plot the components against the time variable.

```
>> [t, y] = ode45(@Lin, [0, 10], [1;1]);
>> plot(t, y(:, 1), 'k');
>> hold on;
>> plot(t, y(:, 2), 'r');
```

We choose the time range $[0, 10]$ and the initial conditions $y_1(0) = y_2(0) = 1$.
Our trajectories are tending to zero. This is not surprising because our matrix is .....???

ODEs

- Of course, the right hand side does not have to be linear.
- Lotka-Volterra systems play an important role in mathematical ecology.
- These take the general form

$$\dot{x}_i = x_i(\sum_{j=1}^{n} a_{ij}x_j + b_i).$$

- Take the same matrix as in our linear example and choose

$$b = \begin{pmatrix} 3 \\ 1 \end{pmatrix}.$$

ODEs

```
function dydt = LV(t, y)

dydt = zeros(2, 1);

dydt(1) = y(1)*(-5*y(1)  + 2*y(2) + 3);
dydt(2) = y(2)*(y(1) - 3*y(2) + 1);

end
```

Once we have defined the function `LV`, we solve it in exactly the same way as before.

```
>> [t, y] = ode45(@LV, [0, 10], [1;1]);
>> plot(t, y(:, 2), 'r');
>> hold on
>> plot(t, y(:, 1), 'k');
```

Graphs

- A standard way of representing graphs is to use the adjacency matrix.
- Here $a_{ij} = 1$ if there is an edge from i to j and $a_{ij} = 0$ otherwise.
- We only consider undirected graphs.
- The classical Erdos Renyi random graph model on n vertices assigns an edge to each pair (i, j) with probability p.
- It is possible to generate ER graphs (and other models) and study their properties in MATLAB.

ER Graphs

Let's write a simple function to generate an ER graph on n vertices with edge probability p.

```
function adj = ERVec(node_count, p)

%generates the adjacency matrix of an Erdos-Renyi random graph
%node_count is the size of the graph; p is the probability of
%a pair of nodes being connected by an edge

adj = zeros(node_count);

Tval = triu(rand(node_count), 1);

I = find(Tval > 1-p);
adj(I) = 1;
adj = adj+adj';
end
```

The function `triu(A, k)` returns the upper triangular part of A starting at the 1st upper diagonal.
The function `find(condition)` returns the indices in `Tval` where the condition is satisfied.

Graphs

- The degree of a node is the number of edges incident with it.
- Two notes are neighbours if they are linked by an edge.
- Write some simple MATLAB functions to calculate: the degree of a node; the neighbours of a node; the maximal, minimal and average degree of a graph.