COMPUTATIONAL MODELING WITH MATLAB

Neslon Mandela African Institute of Science and Technology
Arusha, Tanzania

NSF-USAID PEER 2013 PROGRAM

Padmanabhan Seshaiyer
George Mason University
Fairfax, Virginia, USA
Introduction

MATLAB is an interactive system and programming language for general scientific and technical computation. MATLAB is widely used in many universities in introductory as well as advanced courses in mathematics, science and engineering. The basic data element in MATLAB (MATrix LABoratory) is a matrix that does not require dimensioning. This allows solution of many numerical problems in a fraction of the time it would take to write a program in other high level programming languages such as FORTRAN or C. Furthermore, problem solutions are expressed in MATLAB almost exactly as they are written mathematically.

Perhaps the easiest way to visualize MATLAB is to think of it as a fully featured calculator. Like a basic calculator, it does simple math such as addition, subtraction, multiplication and division. Like a scientific calculator, it handles complex numbers, exponents and square roots, logarithms, and trigonometric operations such as sine, cosine, and tangent. Like a programmable calculator, you can store and retrieve data; you can create, execute and save sequences of commands to automate the computation of important equations; you can make logical comparisons and control the order in which commands are executed. Like the most powerful calculators available, it allows you to plot data in a wide variety of ways, perform matrix algebra, manipulate polynomials, integrate functions, manipulate equations symbolically etc. In reality, MATLAB offers many more features and is more multifaceted than any calculator. MATLAB is a tool for making mathematical calculations. MATLAB is a user-friendly programming language with features more advanced and much easier to use than other high level programming languages. It provides a rich environment for data visualization through its powerful graphic capabilities. MATLAB is an application development platform where sets of intelligent problem solving tools for specific application areas, often called TOOLBOXES, can be developed with relative ease.

Because of the vast power of MATLAB, it is important to start with the basics. That is, rather than throw everything at you and hope that you understand some of it, in the beginning it is helpful to think of MATLAB as a calculator. First as a basic calculator. Next as a scientific calculator. Then as a programmable calculator. And finally as a top-of-the-line calculator. By using this calculator analogy, you will see the ease with which MATLAB can be used in problem solving and computation in a straightforward manner.

These lecture notes are designed to provide an easy introduction to MATLAB. Depending on your background, you may find parts of this lecture notes boring or some of it may be over your head. In either case, I will try to find a point in the notes where you become comfortable to use MATLAB for your computations.

While these notes are not comprehensive, I plan to keep building on the material. So, please contact me on a regular basis for the updated version of these lecture notes.

Good Luck,

Padmanabhan Seshaiyer

Padmanabhan Seshaiyer
FIRST STEPS

You should start MATLAB by simply typing `matlab` if you are working on a Unix system or just by double clicking the MATLAB icon if you are using a Windows based system. If all goes well you will see a MATLAB prompt

```matlab
>>
```
inviting you to initiate a calculation. In what follows, any line beginning with `>>` indicates typed input to MATLAB. You are expected to type what follows by not the `>>` prompt itself. MATLAB supplies that automatically.

**Arithmetic with MATLAB**

MATLAB understands the basic arithmetic operations: +, -, *, /. Powers are indicated with ^, thus typing

```matlab
>> 5*4 + 3^2
```
and pressing enter, results in

ans = 29

The laws of precedence are built in but if in doubt, you should put parentheses appropriately. For example,

```matlab
>> 7+3*(2/(8-2))
```
ans = 8

The sort of elementary functions familiar on hand calculators are also available. For example,

```matlab
>> sqrt(3^2+4*4)
```
ans = 5

```matlab
>> exp(log(3.2))
```
ans = 3.2

One of the most useful commands in MATLAB is the `help` command. For e.g., you can type,

```matlab
>> help sin
```

*Padmanabhan Seshaiyer*
**Using Variables**

You can assign numerical values to variables for use in subsequent calculations. For example the volume of a sphere can be calculated via the following sequence:

```plaintext
>> radius = 3;
>> volume=(4/3)*pi*radius^3
volume =
 113.0973
```

Note that the first line did not seem to produce a result in screen. When MATLAB encounters an instruction followed by a semi-colon ; it suppresses any visual confirmation. It really does obey the instruction and records the value of the variable in the memory. This is useful if you want to avoid cluttering up the screen with intermediate results. Each variable must somehow be assigned before you make use of it in further calculations. For example if you have followed the above example with,

```plaintext
>> x=4*pi*radius*h
```

you should get the result,

```plaintext
??? Undefined function or variable 'h'
```

This is self-explanatory. If you now type

```plaintext
>> h=3;
>> x=4*pi*radius*h
```

you should have more success. Incidentally, a quick way of repeating a previous MATLAB instruction is to press the ‘up-arrow’ key until you recover the command you need. You may also use the sideways arrows to modify any of the previous commands.

At any point typing the command,

```plaintext
>> who
```

tells about all the variables that are in the workspace of the current MATLAB session.

*Padmanabhan Seshaiyer*
Using Vectors

A vector can be input in several ways. For example,

```matlab
>> u=[1 3 5];
>> v=[1,3,5];
>> w=1:2:5;
```

All three commands above yield the same vector. One can perform vector operations such as,

```matlab
>> z=u+v-w;
```

Note that the vectors described so far are row vectors. To get a corresponding column vector, we invoke the transpose of a vector.

For example,

```matlab
>> u'
```

```matlab
ans =
    1
    3
    5
```

The difference between the vectors u and u’ can be understood by typing the commands

```matlab
>> size(u)
>> size(u')
```

which yields the row size followed by the column size for each vector. One can now multiply vectors appropriately to either yield an inner product

```matlab
>> z=u*u'
```

or a matrix,

```matlab
>> z=u' *u
```

Multiplication of vectors only happens if the inner matrix dimensions agree.

For example,

```matlab
>> u*u
```

would yield,

```matlab
??? Error using ==> *
Inner matrix dimensions must agree.
```
Suppose we want a set of values $z$ given by $z = u^2$ then we want

```
>> z = u.*u;
```

where the . is inserted before the * symbol which forces an element-by-element operation. Similarly $u./v$ and $u.^3$ can be understood to be the corresponding element-by-element operations.

**Example:** Suppose we want to find the distance between two points A and B whose position vectors are given by $a = (1, 2)$ and $b = (5, 5)$ respectively. The following sequence of commands can be executed in MATLAB to obtain the desired result:

```
>> a=[1 2];
>> b=[5 5];
>> d = b - a;
>> dd = d*d';
>> dist_ab = sqrt(dd)
```
Using Script Files

Now suppose we want to modify the position vectors and find the distance between the new coordinates, it will not be worthwhile to type all the commands again. Instead one can create a *script* file, which will contain all the necessary information. To create a new script file, one can open their favorite text editor and type the following commands,

```matlab
% dist_ab.m
% Calculates distance between any two position vectors a and b
d=b-a;
dd=d*d';
dist_ab=sqrt(dd)
```

Save these contents in a file named *dist_ab.m* and note that the script file has a ‘.m’ extension which denotes that it is a MATLAB file. Now one can pick values for the vectors *a* and *b* by simply typing say,

```matlab
>> a=[1 2 3];
>> b=[5 5 3];
```

Then find the distance by simply typing

```matlab
>> dist_ab
dist_ab =
    5
```

This program can be called repeatedly to find the distance between any two position vectors that are entered by the user.
Using Function Files

It was tedious to have to assign the two vectors each time before using the above script file. One can combine the assignment of input values with the actual instruction, which invokes the script file by using a function m-file. Not only that, but also one can at the same time assign the answer to an output variable.

To create a new function file, one can open their favorite text editor and type the following commands,

```matlab
% distfn.m
% Calculates the distance between two vectors a and b
% Input: a, b (position vectors)
% Output: dist_ab is the distance between a and b
function dist_ab = distfn(a, b)
    d = b - a;
    dd = d*d';
    dist_ab = sqrt(dd);
```

Save these contents in a file named `distfn.m` and we should now be able to run,

```matlab
>> dist_ab = distfn([1 2 3], [5 5 3])
```
or

```matlab
>> a = [1 2 3];
>> b = [5 5 3];
>> dist_ab = distfn(a, b);
```

To save a certain part of the work in the MATLAB session, one can type,

```matlab
>> diary work1
>> ....
>> diary off
```

All the work between the diary commands will be saved into a file called `work1`. Also note that

```matlab
>> help distfn
```

would return the comments that were entered at the beginning of the program.

**Homework:** Create a function file to deduce the maximum length of the side of a triangle ABC whose position vectors are given by `a`, `b`, `c`.

*Padmanabhan Seshaiyer*
Using Matrices

To define a matrix you start with its name. Square brackets are used to begin and end each matrix. You can separate elements with a space or a comma, and end each row with a semi-colon like this:

```>> A = [1, 2, -6; 7, 5, 2; -2, 1, 0]
```

or this:

```>> A = [1 2 -6; 7 5 2; -2 1 0]
```

In either case you have defined the same 3 by 3 matrix named A and you can use it at any time in a calculation or to define another matrix by using its name (A).

Examples:

```>> B = [2 -4 7]
```

B is a 1 by 3 row matrix

```>> C = [2; -4; 7;]
```

C is a 3 by 1 column matrix

```>> D = [5, 8,-2; 3,-4, 5]
```

D is a 2 by 3 matrix

You can edit individual elements of a matrix as follows:

```>> A(1,1) = -5      changes the element in row1 column1 of matrix A to -5
```

```>> D(2,3) = 0      changes the element in row2 column3 of matrix D to 0
```
To perform operations on matrices you use the symbols ( +, -, * ) from the number keypad or above the numbers across the top of the keyboard.

Examples:

\[
\begin{align*}
&\text{>> A + A} & \text{add the matrix A to itself.} \\
&\text{>> B * C} & \text{multiplies B and C} \\
&\text{>> C*B - A} & \text{A is subtracted from the product of C and B}
\end{align*}
\]

The symbol \(^\wedge\) (above the number 6) is used to raise a matrix to an exponent as follows:

\[
\begin{align*}
&\text{>> A^3} & \text{cubes the matrix A (you might also use A*A*A for the same calculation)} \\
&\text{>> C*D} & \text{an error message is displayed Matlab will give an error message when the calculation cannot be done because of a dimension mismatch.}
\end{align*}
\]

To solve the system of equations for \((x, y, z)\) using Gaussian Elimination:

\[
\begin{align*}
a x + b y + c z &= u \\
e x + f y + g z &= v \\
p x + q y + r z &= w
\end{align*}
\]

we perform the following steps in matlab.

\[
\begin{align*}
&\text{>> A = [a b c; e f g; p q r];} \\
&\text{>> b = [u; v; w];} \\
&\text{>> M = [A b];} \\
&\text{>> R = rref(M);} \\
&\text{>> X = R(4, 1:3);}
\end{align*}
\]

Note that this employs \textit{rref} corresponds to the Reduced row echelon form.
**Plotting Basics in 2D**

Suppose we wish to plot the graph $y=x^2$ in the interval [-1, 1], then just type,

```matlab
>> x = -1:0.1:1
>> y=x.*x
>> plot(x,y)
```

Note that the axes are automatically chosen to suit the range of variables used. One can add more features to the plot using the following commands:

```matlab
>> title('Graph of y=x^2')
>> xlabel('x')
>> ylabel('y')
```

Suppose now we want to plot the graphs $y_1=x^2$ and $y_2=2x$ on the same figure, we need,

```matlab
>> y1=x.*x;
>> y2=2*x;
>> plot(x,y1)
>> hold on
>> plot(x,y2,'ro')
```

Note that the `hold on` command tells MATLAB to retain the most recent graph so that a new graph can be plotted. Note that axes are adjusted and the second curve is plotted with a red circle. More options on the choice for the color and symbol can be found by typing,

```matlab
>> help plot
```

Suppose we want to plot the graphs $y_1=x^2$ and $y_2=2x$ on the same figure but partitioned into two subplots then we say,

```matlab
>> subplot(2,1,1)
>> plot(x,y1)
>> subplot(2,1,2)
>> plot(x,y2)
```

One must now be able to employ the various plotting commands in various combinations to get the desired figure in 2D.
Plotting Basics in 3D

Three-dimensional (3D) plots can be a useful way to present data or functions of multiple variables. MATLAB provides several options to displaying 3D data.

To plot the parametric equations \( x(t) = t \sin(3t), y(t) = t \cos(3t), z = \sqrt{t} \) in \( 0 \leq t \leq 4\pi \) a 3D line plot is created as follows:

```matlab
>> t=0:0.1:4*pi;
>> x=t.*sin(3*t);
>> y=t.*cos(3*t);
>> z=sqrt(t);
```

Once the \( x, y, z \) are entered, to create the basic 3D line plot we use the "plot3" command.

```matlab
>> plot3(x,y,z,'k','linewidth',1)
>> grid on
>> xlabel('x'); ylabel('y'); zlabel('z')
```

These set of commands should created the following 3D line plot.
To create a mesh or surface plots for the function \( z = f(x, y) = \sqrt{x^2 + y^2} \sin(x) \cos(y) \) over \( -2 \leq x, y \leq -2 \), we can use the following set of commands:

```matlab
>> x=-2:0.1:2;
>> y=-2:0.1:2;
```

Once we have the x, y as coordinates in vector forms, we can use MATLAB's built-in function, "meshgrid" to create two matrices X, Y that correspond to the x-coordinates and y-coordinates of the grid points respectively.

```matlab
>> [X,Y]=meshgrid(x,y);
```

Note that in the vectors x and y, the first and last elements are the respective boundaries of the domain and the density of the grid will depend on the number of elements in the vectors. Once we have these matrices X and Y, we can then evaluate the value of z at each point by using component wise calculation. This is done as follows:

```matlab
>> Z = sqrt(X.^2+Y.^2).*sin(X).*cos(Y);
```

To illustrate the output using two different approaches, we are plotting the output using both the "mesh" and the "surf" commands.

```matlab
>> subplot(1,2,1)
>> mesh(X,Y,Z)
>> xlabel('x'); ylabel('y'); zlabel('z')
>> subplot(1,2,2)
>> surf(X,Y,Z)
>> xlabel('x'); ylabel('y'); zlabel('z')
```
More helpful MATLAB commands:

quit or exit either of these closes the program and ends your matlab session.

save filename this command will save all variables (and ONLY the variables - NOT the whole session) that you have defined in the current session. This is helpful when you enter large matrices that you want to work with at a later date.

load filename this command loads the variables that you saved in a previous session.

lpr this is the command used to print your diary file. EXIT THE MATLAB PROGRAM.

who displays variables you have defined in your current session.

why matlab answers the question.(again and again)

clear clears all variables from your current session. Only use this command if you want to lose everything.

% this is used for comments. Matlab ignores any line that begins with %

Matlab has many built in matrix functions and operators. I have listed some here that you may find useful:

>> size(A) gives the dimension of the matrix A

>> inv(A) calculates the inverse of the matrix A, if it exists.

>> det(A) calculates the determinant of the matrix A

>> rref(A) calculates the row reduced echelon form of the matrix A

>> A' forms the transpose of the matrix A.

>> eye (2,2) this is the 2 by 2 identity

>> zeros (3,3) builds the zero matrix of any dimension

>> ones (3,2) fills a matrix of any size with ones.

>> rats(A) displays elements of the matrix A as fractions

>> format long displays all numbers with 15 digits instead of the usual 4 digits

Padmanabhan Seshaiyer
Some useful built-in mathematical functions:

*Elementary trigonometric functions and their inverses*
sin, cos, tan, sec, csc, cot, asin, acos, atan, asec, acsc, acot

*Elementary hyperbolic functions and their inverses*
sinh, cosh, tanh, sech, csch, coth, asinh, acosh, atanh, asech, acsch, acoth

*Basic logarithmic and exponentiation functions*
log, log2, log10, exp, sqrt, pow

*Basic Statistical functions*
max, mean, min, median, std, var, sum

*Basic complex number functions*
imag, real, i, j, abs, angle, cart2pol

*Basic data analysis functions*
fft, ifft, interpn, spline, diff, del2, gradient

*Basic logical functions*
and, or, xor, not, any, all, isempty, is*

*Basic polynomial operations*
poly, roots, residue, polyfit, polyval, conv, deconv, polyder

*Basic plotting functions*
okit, axis, grid, semilogx, semilogy, loglog, bar, barh, stairs, stem, pie, hist, polar, subplot, mesh,
surf, surfl, waterfall, contour, sphere, cylinder, plot3, bar3, stem3, scatter3, pie3

*Function functions that allows users to manipulate mathematical expressions*
feval, fminbnd, fzero, quad, ode23, ode45, vectorize, inline, fplot, explot

*Basic matrix functions*
zeros, ones, det, trace, norm, eig

Try the following:
```
>> f='2+cos(x)';
>> f=inline(f);
>> fplot(f,[0,2*pi])
>> x=fzero(inline('cos(x)','x'),1)
```
for Loops
This allows a group of commands to be repeated a fixed, predetermined number of times. The
general form of a for loop is:
    for x = array
        Commands ...
    End
For example,
>> for n=1:10
    x(n)=sin(n*pi/10);
end
yields the vector x given by,
>> x
x =
 Columns 1 through 7
0.3090  0.5878  0.8090  0.9511  1.0000  0.9511  0.8090
 Columns 8 through 10
0.5878  0.3090  0.0000
Let us now use the for loop to generate the first 15 Fibonacci numbers 1,1,2,3,5,8,…
>> f=[1 1];
>> for k=1:15
    f(k+2) = f(k+1) + f(k);
end
>> f
**while Loops**

This evaluates a group of commands an infinite number of times unlike the `for` loop that evaluates a group of commands a fixed number of times. The general form for while loop is

```
while expression
    Commands ...
end
```

For example to generate all the Fibonacci numbers less than 1000, we can do the following:

```matlab
>> f=[1 1];
>> k=1;
>> while f(k) < 1000
    f(k+1) = f(k+1) + f(k);
    k = k +1;
end
>> f
```
**if-else-end Construct**

There are times when a sequence of commands must be conditionally evaluated based on a relational test. This is done by the if-else-end construct whose general form is,

```
if expression
    Commands ...
end
```

For example if we want to give 20% discount for larger purchases of oranges, we say,

```
>> oranges=10;             % number of oranges
>> cost = oranges*25       % Cost of oranges
cost =
    250
>> if oranges > 5
    cost = (1-20/100)*cost;
    end
>> cost
cost =
    200
```

If there are more conditions to be evaluated then one uses the more general if-else-end construct given by,

```
if expression
    Commands evaluated if True
else
    Commands evaluated if False
end
```
**switch-case Construct**

This is used when sequences of commands must be conditionally evaluated based on repeated use of an equality test with one common argument. In general the form is,

```
 switched expression
   case test_expression1
       Commands_1 ...
   case test_expression2
       Commands_2 ...
   ....
   otherwise
       Commands_n ...
 end
```

Let us now consider the problem of converting entry in given units to centimeters.

```matlab
function y=centimeter(A,units)
switch units   % convert A to cms
    case {'inch','in'}
        y = A*2.54;
    case {'feet','ft'}
        y = A*2.54*12;
    case {'meter','m'}
        y = A*100;
    case {'centimeter','cm'}
        y = A;
    otherwise
        disp(['Unknown units: ' units])
        y = nan;  %% stands for not a number
end
```

*Padmanabhan Seshaiyer*
These are arrays of ASCII values that are displayed as their character string representation, e.g.:

```plaintext
>> t = 'Hello'
t =
Hello
>> size(t)
ans =
    1     5
```

A character string is simply text surrounded by single quotes. Each character in a string is one element in the array. To see the underlying ASCII representation of a character string, type:

```plaintext
>> double(t)
ans =
     104    101    108    108    111
>> char(t)
ans =
    hello
```

Note that the function char provides the reverse transformation. Since strings are numerical arrays with special attributes, they can be manipulated just like vectors or matrices. For example,

```plaintext
>> u=t(2:4)
u =
ell
>> u=t(5:-1:1)
u =
olleh
>> u=t(2:4)'
u =
e
l
```
One can also concatenate strings directly. For instance,

```matlab
>> u='My name is ';
>> v='Mr. MATLAB';
>> w=[u v]
w =
    My name is Mr. MATLAB
```
The function `disp` allows you to display a string without printing its variable name.

```matlab
>> disp(w)
My name is Mr. MATLAB
```
In many situations, it is desirable to embed a numerical result within a string. The following string conversion performs this task.

```matlab
>> radius=10; volume=(4/3)*pi*radius^3;
>> t=['A sphere of radius ' num2str(radius) ' has volume... of ' num2str(volume) '.'];
>> disp(t)
```

It may sometimes be required to find a certain part of a longer string. For example,

```matlab
>>a='Texas Tech University';
>>findstr(a,' ')   %% Finds spaces
ans =
    6  6 11
>>findstr(a,'Tech')  %% Finds the string Tech
ans
    7
>>findstr(a,'Te')  %% Finds the string starting with Te
ans =
    1    7
>>findstr(a,'tech')  %% This command is case-sensitive
ans =
    [  ]
```
If it is desired to replace all the case on Tech to TECH then one can do this by using,

```matlab
>>strrep(a,'Tech','TECH')
ans =
```

Padmanabhan Seshaiyer
OPERATORS

Relational Operators
These include the operators $<$  $<=$  $>$  $>=$  $==$  $~=$
These operators can be used to compare two arrays of the same size, or to compare an array to a scalar. For example:

```
>> A=1:5, B=5-A
A =
    1  2  3  4  5
B =
    4  3  2  1  0
>> g=A>3
g =
    0  0  0  1  1
```
finds elements of A that are greater than 3. Zeros appear in the result where this condition is not satisfied and ones appear where it is.

```
>> for n=2:6
    if rem(n,2)~=0
        n
    end
end
n =
    3
n =
    5
```
**Logical Operators:**

The three main logical operators are: AND, OR, NOT which are represented by the symbols &, |, ~ respectively in MATLAB. Often one can combine these operators with relational expressions. For example:

```matlab
>> g=(A>2) & (A<5)
g =
    0     0     1     1     0
```

returns ones where A is greater than 2 AND less than 5.

```matlab
>> for n=2:6
    if (rem(n,2)~=0) & (rem(n,3)~=0)
        n
    end
end
n =
    5
```

Finally, the above capabilities make it easy to generate arrays representing discontinuous functions, which are very useful in understanding signals etc. The basic idea is to multiply those values in an array that you wish to keep with ones, and multiply all other values with zeros. For example,

```matlab
>> x = -2:0.1:2;  % Creates Data
>> y = ones(size(x));
>> y = (x>-1) & (x<1);
>> plot(x,y)
```

This should plot a function which resembles part of a square wave that is discontinuous at x=-1 and x=1.
MODEL EXAMPLE: The Monte Carlo Method

This is a technique to simulate an analytic model of a phenomenon where the computer is instructed to randomly try various inputs and to tabulate the results. This statistical data is often enough to answer the question posed. Let us consider a “dartboard” integration method for computing integrals via the Monte Carlo method. Let us assume our goal is to compute the area below the curve $y=f(x)$. The idea is to pick a rectangular region based on the limits of integration and start throwing darts at this region. The integral is the success rate times the target area. Hence by repeatedly choosing a point at random within the rectangle enclosing the curve, we keep count of the number of times this random point falls under the curve (success). After a large number of experiments are performed, we multiply the area of the enclosing rectangle by the success rate to obtain an estimate of the area under the curve.

Let us estimate the value of the integral of the function $f(x)=\exp(-x^3)$ between $x=0$ and $x=1$. For this once can draw the box $0 \leq x \leq 1$ and $0 \leq y \leq 1$ to enclose the curve. We now exploit the built-in random number generator `rand` of MATLAB that will generate uniformly distributed random numbers between 0 and 1:

```matlab
% mcint.m
A = 1; % area of enclosing rectangle
N=10000; % set the number of trials
s=0; % initialize success counter
for i=1:N % begin trials
    x = rand; % choose a random x-coordinate
    y = rand; % choose a random y-coordinate
    if y <=exp(-x^3) % if below the curve, then
        s = s + 1; % increment successes
    end; % end if
end; % end of each trial
I=A*s/N % integral = area * success/trials
```

Padmanabhan Seshaiyer
The routine yields an estimate of $I = 0.805$. In truth, for functions of one variable, there are superior methods of numerical integration, such as trapezoid, Simpson’s rule etc. Nevertheless, important and practical uses of Monte Carlo often involve the evaluation of difficult integrals for which there is no explicit analytic model.

**Homework:**

1. Find the area between the curves $y=x^3$ and $y=x^2-x$ between $x=0$ and $x=1$ by (a) Monte Carlo integration method and; (b) the Trapezoid rule. Compare your answers in parts (a) and (b) with the exact answer obtained by direct integration.
**MODEL EXAMPLE: Discrete-time Age-structure Models**

The basic idea for developing these models is to take a vector representing the age distribution in one year. Then construct a matrix of transition probabilities from one year to the next and finally use matrix multiplication to predict the probable age distribution for the next year and other subsequent years.

**Example**: Suppose we count number of people in age-bands 0-5, 6-19, 20-59, 60-69. We will make the assumptions that

- Within one age-band, the age distribution is constant;
- We do not consider anyone who lives beyond 69;
- Deaths only occur in the 60-69 age-band at the rate of $d_A\%$ per annum and in the 0-5 band at $d_I\%$ per annum;
- Births are due to people in the 20-59 age-band at the rate of $b\%$ per annum.

Our objective is to relate population in the four age-bands in year 2 to year 1. This data can then be captured as follows:

<table>
<thead>
<tr>
<th>Age Band</th>
<th>Number in Band</th>
<th>Year Groups Die that Year Survive that Year Leave Band Enter Band</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5</td>
<td>$n_1$</td>
<td>6</td>
</tr>
<tr>
<td>6-19</td>
<td>$n_2$</td>
<td>14</td>
</tr>
<tr>
<td>20-59</td>
<td>$n_3$</td>
<td>40</td>
</tr>
<tr>
<td>60-69</td>
<td>$n_4$</td>
<td>10</td>
</tr>
</tbody>
</table>
From this data the relationship between age-band in consecutive years can be summarized as:

\[
\begin{bmatrix}
    n_1(t+1) \\
    n_2(t+1) \\
    n_3(t+1) \\
    n_4(t+1)
\end{bmatrix} =
\begin{bmatrix}
    \frac{5}{6} \left(1 - \frac{d_t}{100}\right) & 0 & \frac{b}{100} & 0 \\
    \frac{1}{6} \left(1 - \frac{d_t}{100}\right) & \frac{13}{14} & 0 & 0 \\
    0 & \frac{1}{14} & \frac{39}{40} & 0 \\
    0 & 0 & \frac{1}{40} & \frac{9}{10} \left(1 - \frac{d_A}{100}\right)
\end{bmatrix}
\begin{bmatrix}
    n_1(t) \\
    n_2(t) \\
    n_3(t) \\
    n_4(t)
\end{bmatrix}
\]

which can be simplified using \( N(t) = \begin{bmatrix} n_1(t) \\ n_2(t) \\ n_3(t) \\ n_4(t) \end{bmatrix} \) which yields \( N(t+1) = L N(t) \) and \( N(t) = L' N(0) \).

The following program constructs the Leslie matrix \( L \) to model the behavior of the populations age structure using the age bands.

```matlab
% leslie.m
% Initialization
disp('Leslie matrix model for population age structure')
disp('')
disp(' the age bands 0-5, 6-19, 20-59, 60-69.')
disp('You will be prompted for birth and death rates etc.')
b = input('Birth rate as a percentage of 20-59 band /yr?: ');
di = input('Infant death rate as a percentage of 0-5 band /yr?: ');
da = input('Age-related death rate as a percentage of 60-69 band /yr?: ');

%Set up Leslie matrix
b = b/100; di = di/100; da = da/100;
disp(' Leslie matrix is:')
L = [ 5*(1-di)/6 0 b 0; 
     (1-di)/6 13/14 0 0; 
     0 1/14 39/40 0; 
     0 0 1/40 9*(1-da)/10 ]
```

Padmanabhan Seshaiyer
MODEL EXAMPLE: Discrete-time nonlinear population dynamics

Consider a strain of bacteria that are growing in a media containing sufficient nutrients doubles in size and divide every \( t \) minutes. Suppose we begin with a known biomass (dry weight) of cells, say \( P_0 \), at time zero and monitor the population size at successive division times. Then the population of the cells after \( t \) minutes can be related to the initial population as \( P_1 = 2P_0 \). This can be generalized as \( P_{n+1} = 2P_n \) which describes the population measurements when there is adequate nutrients and when the sampling and double times are both \( t \) minutes. When the biomass of cells becomes large, competition for the limited amount of nutrients ensues, and the bacteria get an inadequate supply to sustain division every \( t \) minutes. When the population is large, it less than reproduces itself each interval. This can be described as: 

\[
P_{n+1} = \frac{2P_n}{1 + \frac{P_n}{C}}
\]

where \( C \) is a saturation constant. Note that when \( P_n = C \), reproduction is only one-half its maximum. If we are interested in finding the population at steady-state then we are interested in solving the nonlinear equation: \( x = \frac{2x}{1 + \frac{x}{C}} \). Denoting \( g(x) = \frac{2x}{1 + \frac{x}{C}} \) we can solve this by finding the fixed-point of the equation \( x = g(x) \). This algorithm is implemented below.

```matlab
function x = fixed_point(g,x0,niter)
    x(1) = x0;
    for i = 1: niter
        x(i+1) = g(x(i));
    end
    x = x(i+1);
end
```

To execute this at the MATLAB prompt, one can run this with initial guess = 10 and \( C = 50 \),

\[
g = \text{inline}(2x/(1 + x/50))
\]

\[
g(x) = \frac{2x}{1 + \frac{x}{50}}
\]

\[
\text{fixed_point}(g,10,10)
\]

ans = 49.8054

Incidentally, it should be clear by solving the equations or drawing a cob-web diagram why the answer is converging to \( C \).

Padmanabhan Seshaiyer
Let us try to enhance the code by drawing a cob-web diagram.

```matlab
% Program: Fixed Point Method with cob-web diagram
fn=inline('2*x./(1 + x./50)','x');
fxpt=inline('2*x./(1 + x./50)-x','x');
fzero(fxpt,25)
xmin=0;
xmax=100;
N=15;
dx=(xmax-xmin)/10;
x=xmin:dx:xmax;
y=fn(x);
plot(x,y,'b')
set(gca,'FontSize',18)
hold on
plot([xmin xmax],[xmin xmax],'g');
x0=.85;
Y=[x0];
xx=x0;
for i=1:N
    yy=fn(xx);
    Y=[Y yy];
    xx=yy;
end;
YY(1)=Y(1);
for i=1:N
    XX(2*i-1)=Y(i);
    XX(2*i)=Y(i);
    YY(2*i)=Y(i+1);
    YY(2*i+1)=Y(i+1);
end;
XX(2*N+1)=Y(N+1);
plot(XX,YY,'r','LineWidth',1.5);
axis([xmin xmax min(y)-.1 max(y)+.1])
grid on
hold off
```

The cob-web plot is shown in the next page.
Suppose we are interested in fitting a polynomial $P(x) = a_0 + a_1x + a_2x^2 + .... + a_nx^n$ to a given set of data $(x_i, Y_i)$ for $i = 1..M$, (where $M > n$) then we want to find the coefficients $a_i$ such that:

$$
Y_1 = a_0 + a_1x_1 + a_2x_1^2 + .... + a_nx_1^n
$$

$$
Y_2 = a_0 + a_1x_2 + a_2x_2^2 + .... + a_nx_2^n
$$

.....

$$
Y_M = a_0 + a_1x_M + a_2x_M^2 + .... + a_nx_M^n
$$

which implies

$$
\begin{bmatrix}
Y_1 \\
Y_2 \\
\vdots \\
Y_M
\end{bmatrix}_{M \times 1} =
\begin{bmatrix}
1 & x_1^2 & \ldots & x_1^n \\
1 & x_2^2 & \ldots & x_2^n \\
\vdots & \vdots & \ddots & \vdots \\
1 & x_M^2 & \ldots & x_M^n
\end{bmatrix}_{M \times n} \begin{bmatrix} a_1 \\
a_2 \\
\vdots \\
a_n \end{bmatrix}_{n \times 1}
$$

In vector form this becomes $\vec{a} = \vec{b}$ which can be solved for the coefficients by solving the associated normal equations $A^T A \vec{a} = A^T \vec{b}$. This can be implemented in two ways in MATLAB. In the first method we construct the matrix from the data points $x_i$ and solve for the coefficients and plot them as follows.

```matlab
% Fitting an nth degree polynomial to given data

function a=polynomial_fit(x,y,n)
    A = ones(size(x))';
    for i = 1:n
        A = [A x.^i];
    end
    a = (A'*A)
    Y = a(1);
    xg=x(1):0.1:x(length(x));
    for i=2:n+1
        Y = Y + a(i)*xg.^(i-1);
    end
    plot(x,y,'ro',xg,Y);
    title('Best-fit curve')
    xlabel('x')
    ylabel('Y')
```

Note that method 2 is to directly use a built-in command polyfit(x,y,n) with n specified yields the same results. While method 2 maybe simple, one has to resort to method 1 if we were to fit a nonlinear relationship between data sets x and y.

*Padmanabhan Seshaiyer*
One can also take advantage of the MATLAB built-in basic fitting interface tool that can be found under the "tools" when plotting a figure. By using this interface the user can fit a curve to the data points with polynomials of various degrees as well as with spline and Hermite interpolation methods. It also allows to plot various fits on the same plot that helps to compare the fits. Other features include plotting the residuals of the various polynomial fits and compare them to the norm of the residuals, calculating the value of the fit at specific points. Finally, the user also has the opportunity to add other polynomial equations to plot.

The basic fitting window and the associated plots for a given data is displayed below:
MODEL EXAMPLE: Solving Initial value problems

Consider solving in the initial value problem: \[ \frac{dy}{dt} = 2t \, y \]
\[ y(0) = 1 \]

The exact solution to the problem can be easily worked out to be \( y(t) = e^{t^2} \). The Euler's method for this problem becomes: \( y_{i+1} = y_i + h \, f(t_i, y_i) \) for \( i = 1 \ldots n \) and where \( f(t_i, y_i) = 2t \, y_i \).

In the MATLAB code below we will implement the Euler's method and plot the approximate solution against the exact solution.

% Myeulers.m
% This is the Eulers Method for the equation \( y'(t) = 2ty, y(0)=1 \)
t(1)=0; % Initial t
y(1)=1; % Initial y
h = 0.1; % Step size
N = 11; % Nodes
for i=1:N-1
y(i+1)=y(i)+h*2*t(i)*y(i); % y increment
t(i+1)=t(i)+h; % t increment
end
plot(t,y,'.') % plot approximate solution
hold on % Hold on
%% Plot the exact solution
t1=0:0.01:1;
y1=exp(t1.*t1);
plot(t1, y1,'r')
legend('Approximate', 'Exact')
xlabel('t')
ylabel('y(t)')

MATLAB already has several built-in solvers for solving initial value problems. One may use any of these solvers but the one that is recommended is the Runge-Kutta method that is built-in as "ode45". The basic syntax for using ode45 for solving \( y' = yprime(t, y) \)

\[ [t, x] = \text{ode45('yprime',} [t_o, t_f ],y_0); \]
where $y_0$ is the initial condition and the problem is solved on the interval $t_o \leq t \leq t_f$. For the given problem, we can use:

$$[t, y] = \text{ode45('test', [0,1],1);}$$

where test.m is the M-file which contains the definition of the equation, i.e.,

```matlab
function yprime = test(t, y)
    yprime = 2*t*y;
```

In the plot below we graph the exact and compare it to the Eulers and the Runge-Kutta solutions. Clearly, the Runge-Kutta approximates the solution better which is no big surprise because of its higher order accuracy.

The advantage and simplicity of using ode45 becomes obvious when solving system of first order differential equations. Consider solving the system:

$$
\begin{align*}
  y'_1 &= y_2 - y_1^2 \\
  y'_2 &= -y_1 - 2y_1y_2
\end{align*}
$$

numerically on the interval [0, 10] with $y_1(0) = 0$ and $y_2(0) = 1$.

Padmanabhan Seshaiyer
To do this we first need to describe the system in a derivative M-file. For this we will use the vector $x$ in MATLAB to denote the solution. The first component, which in MATLAB is denoted by $y(1)$, will correspond to $y_1$ and the second component denoted by $y(2) = y_2$. For the derivatives, we will use the vector $yprime$, with the first component corresponding to $y'_1$ and the second component to $y'_2$. Then our system of differential equations can be entered into an M-file as follows:

```matlab
function yprime = test2(t, y)
    yprime(1,1) = y(2) - y(1)^2;
    yprime(2,1) = -y(1) - 2*y(1)*y(2);
end
```

Now to solve the system as before: $[t, x] = \text{ode45}('test2', [0, 10],[0,1]);$

Also we can plot the solutions using

```matlab
>> plot(t,x(:,1),'o',t,x(:,2))
>> legend('y_1','y_2')
>> xlabel('t')
>> ylabel('Solutions')
```

The resulting plot looks like this.

![Plot of solutions](image)
It is also possible to plot the components against each other with the command:

```matlab
>> plot(x(:, 1), x(:, 2));
```

This gives the following *phase plane* plot.

Note that one can also use the ode45 function as:

```matlab
>> [t y]=ode45(@yprime,[t0 tf],y0)
```

The LHS tells Matlab to store the output from ode45 in two vectors, t and y. The arguments on the RHS are:

- `@yprime`: handle for Matlab function returning the value of (1)
- `[t0  tf]`: Beginning and ending time for the desired solution
- `y0`: Initial condition, i.e. $y(t0)$
MODEL EXAMPLE: Solving Boundary value problems

Consider the displacement $u(x)$ of a linear elastic bar under a tangential force $f(x)$. Let the bar be fixed at both ends. One can derive the model for this by invoking Hooke's law that relates the stress to the strain and Newton's law to balance the force. This then yields a boundary value problem that must be solved.

\[ \sigma \alpha \frac{du}{dx} \Rightarrow \sigma = K \frac{du}{dx} \]

\[ \frac{d\sigma}{dx} = -f(x) \]

\[ -\frac{d}{dx}\left(K\frac{du}{dx}\right) = f(x) \]

\[ u(a) = 0 \]
\[ u(b) = 0 \]

For simplicity let us take $K=1$ and $f(x) = (6 + 4x^2)xe^{x^2}$ with the boundary conditions $u(0)=0$, $u(1)=e$. The exact solution for this can be easily verified to be $u(x) = xe^{x^2}$.

This model is implemented via the finite difference method in the MATLAB code below.

```matlab
% myfd.m
% This is a finite difference code
% u_xx = (6 + 4x^2)*x*e^(x^2), u(0)=0, u(1)=e
% Input: a, b, N
function myfd(a,b,N)
    h = (b - a)/N; % Mesh step size
    x = a+h:h:b-h; % Creates the x-grid
    v = h^2*(6 + 4*x.^2).*x.*exp(x.^2); % RHS Vector
    v(N-1) = v(N-1) - exp(1); % Right Boundary cond
    % Create the matrix A
```

Padmanabhan Seshaiyer
A = diag(-2*ones(N-1,1),0) + diag(ones(N-2,1),1);
A = A + diag(ones(N-2,1),-1);
y = A\v'; % Solving the system
yfd = [0 y' exp(1)]

The comparison of the exact and approximate solution for N=10 nodes is shown below.