CITS2401
Computer Analysis & Visualisation

Week 4: Lectures 7 and 8
Matlab functions and script files

Material from MATLAB for Engineers, Moore, Chapter 6-8
Additional material by Peter Kovesi.
Functions

◊ We have already used a number of functions without really examining what’s going on.

◊ In mathematics, the term function means is some operation which is defined over a domain of values such that for any input there is a unique result.

◊ In computer science, the term function has a similar meaning, though we tend to follow a rather general interpretation.

◊ A function is simply a piece of code that receives some data, using the data to return one or more new values to the calling program.

◊ Calling a function - "call up" a specified piece of code to do a particular operation.

◊ The data that is passed to a function is known as the function's arguments (or operands or parameters).
Built-in Functions

◊ Matlab functions can take zero, one, or more arguments, and can return a scalar or a matrix.

◊ For example,

```matlab
>> s=sin(x)

>> value = input('Enter a number ');

>> m = zeros(n, m);

>> [rows, cols] = size(m);
```

◊ Standard Math functions
◊ Special matrix construction functions
◊ Data analysis functions
Examples

◊ For example:

```matlab
>> s = sin(a);
```

- The `sin` function takes a single argument, an angle in radians, and returns to the calling program a single value, the sine of the angle.

◊ Another (non-mathematical) function we have used is the input function:

```matlab
>> value = input('Enter a number ');
```

- The input function takes a single argument, a string of characters, prints it on the screen and waits for the user to type something at the keyboard.
- Whatever the user types at the keyboard is treated as a Matlab expression. If the input is valid, the expression is evaluated and the result is assigned to the variable `value`. 
Examples

◊ Some functions return an array. For example:

```plaintext
>> [rows, cols] = size(m);
```

• The size function takes a single argument, a matrix, and returns two values: the number of rows and columns in the matrix.
Examples

◊ Functions can take more than one argument. For example:

\[
>> m = \text{zeros}(n, m);
\]

• The 
\text{zeros} function takes two arguments, \( n \) and \( m \), and returns a \( n \times m \) matrix of zeroes.

◊ Some functions do not require any arguments. For example:

\[
>> r = \text{rand};
\]

• The \text{rand} function, can accept zero (or more) arguments and returns a pseudo-random number.
Maths Functions

◊ $\sin(a)$ Returns the sine of $a$.
◊ $\cos(a)$ Returns the cosine of $a$.
◊ $\tan(a)$ Returns the tangent of $a$.
◊ $\arcsin(s)$ Returns the inverse sine of $s$.
◊ $\arccos(s)$ Returns the inverse cosine of $s$.
◊ $\arctan(s)$ Returns the inverse tangent of $s$.
◊ $\arctan2(y, x)$ Calculates the inverse tangent of $y/x$, but uses the separate values of $y$ and $x$ to resolve the correct quadrant that the answer lies in. Also deals with cases where $x$ is zero.
More Maths functions:

• abs(a) - Returns the magnitude of $a$.
• mod(a, b) - Returns the remainder of dividing $a$ by $b$.
• round(a) – Rounds a to nearest integer closest to $a$.
• fix(a) - Rounds a to nearest integer that is closest to zero.
• ceil(a) – Rounds a to nearest integer that is closest to positive infinity.
• floor(a) Rounds a to nearest integer that is closest to negative infinity.
• sqrt(v) - Returns the square root of $v$.
• exp(v) - Returns $e^v$.
• log(v) - Returns the natural logarithm of $v$.
• log2(v) - Returns the logarithm to base 2 of $v$.
• log10(v) - Returns the logarithm to base 10 of $v$.

◊ For a full list of the "elementary" functions, type ‘help elfun’ at the Matlab command prompt.
Special matrix construction functions

◊ Matlab provides a number of matrix construction functions.

◊ Some examples include:

•  `zeros(n)` - Returns a $n \times n$ matrix of zeroes.

•  `ones(n)` - Returns a $n \times n$ matrix of ones.

•  `eye(n)` - Returns a $n \times n$ identity matrix.

```
>> eye(3)
ans =
     1     0     0
     0     1     0
     0     0     1
```
Matrix construction functions

◊ Each of these matrix construction functions may accept one or two arguments.

◊ If the function receives two arguments, the arguments specify the number of rows and columns of the matrix separately. For example,

\[
\text{>> ones}(n, m) \quad \% \text{ Generates a } n \times m \text{ matrix of ones.}
\]

◊ You can also have a function call, or expression, as an argument to a function. For example, constructing a matrix of zeroes the same size as an existing matrix:

\[
\text{>> zeros(size(arr));}
\]

• Firstly, the function call `size(arr)` is evaluated. This returns the number of rows and columns in `arr`. The calculated size is then passed to the `zeros` function.
Matrix construction functions

◊ A common use of Matrix construction functions is to pre-allocate memory for matrices prior to a computation. For example,

- Create a new matrix with values equal to twice the values of a given matrix \( m \):

\[
\begin{align*}
\text{[nrows, ncols]} & = \text{size(m)}; \quad \% \text{Find the size of m.} \\
\text{mcopy} & = \text{zeros(nrows, ncols)}; \quad \% \text{Create a matrix to put the answer in.} \\
\text{for } r = 1 : \text{nrows} \\
& \quad \text{for } c = 1 : \text{ncols} \\
& \quad \quad \text{mcopy}(r, c) = 2*\text{m}(r, c); \\
& \quad \text{end} \\
& \text{end}
\end{align*}
\]

◊ What happens if you don’t include the first two lines? How will it affect performance for very large matrices?
Why pre-allocating?

◊ Each time an array is extended, Matlab has to:
  • Create a new array
  • Copy the contents of the old array to the new longer array
  • Add the new values to the array, and then
  • Delete the old array

◊ This process is very time-consuming for long arrays.

◊ A good programming practice is to pre-allocate an array to its maximum size.
Data analysis functions

◊ Matlab provides a number of data analysis functions, such as:

• max(a) - Returns the maximum value in \( a \).
• min(a) - Returns the minimum value in \( a \).
• mean(a) - Returns the mean value in \( a \).
• median(a) - Returns the median value in \( a \).
• std(a) - Returns the standard deviation of the elements in \( a \).
• sort(a) - Returns the elements in \( a \) sorted in ascending order.
• sum(a) - Returns the sum of the elements in \( a \).
• hist(a) - Returns a histogram of the frequency of the elements in \( a \).
General rules on Data Analysis Functions

◊ All of these functions follow the following rules:

• If the argument $a$ is a 1D vector, the result will be a single number.

• If the argument $a$ is a 2D matrix, the specified operation is \textit{applied separately to each column of the matrix}. The results are returned as an array of values.
Examples

◊ For example:

```matlab
>> a = [1 2 3
      4 5 6
      7 8 9];

>> min(a)
ans =
   1 2 3

>> median(a)
ans =
   4 5 6

>> sum(a)
ans =
  12 15 18
```
Examples – one step further

◊ To find the minimum value within a matrix we need to apply the min function twice:

    >> min(min(a))      % The second call to min finds
    ans =              % the minimum value in the array
          1            % obtained from the first call
          % to min.

◊ Similarly, to find the sum of all the values in a matrix we need to apply the sum function twice:

    >> sum(sum(a))     % First call to sum returns:
    ans =             % [7 11 16; 17 21 26; 27 31 36]
          45

◊ **Try** `sum(a(:))`

◊ Familiarity with these functions will save you from having to write many `for` loops.

◊ For more information, type ‘`help datafun`’ at the Matlab command prompt.
### Variance and Standard Deviation

- **std(x)** $\sigma$
- **var(x)** $\sigma^2$

\[
\sigma^2 = \frac{\sum_{k=1}^{N} (x_k - \mu)^2}{N - 1}
\]

**Standard Deviation**
3.6 Random Numbers

◊ **rand(x)**
  - Returns an x by x matrix of random numbers between 0 and 1

◊ **rand(n,m)**
  - Returns an n by m matrix of random numbers

◊ These random numbers are evenly distributed
The University of Western Australia

MATLAB 7.12.0 (R2011a)

Command Window

```matlab
>> rand(3)
ans =
    0.8147    0.9134    0.2785
    0.9058    0.6324    0.5469
    0.1270    0.0975    0.9575
```

```matlab
fx >>
```

Workspace

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>[2, 5, 6]</td>
</tr>
<tr>
<td>ans</td>
<td>[0.8147, 0.9134, 0.2785]</td>
</tr>
<tr>
<td>b</td>
<td>[2, 1, 2]</td>
</tr>
<tr>
<td>g</td>
<td>0.9536</td>
</tr>
<tr>
<td>x</td>
<td>[1, 5, 3; 2, 4, 6]</td>
</tr>
</tbody>
</table>

Command History

```matlab
sortrows(x, 2)
```

```matlab
sortrows(x, -2)
```

```matlab
clc
```

```matlab
size(x)
```

```matlab
length(x)
```

```matlab
numel(x)
```

```matlab
clc
```

```matlab
rand(3)
```
```matlab
>> rand(3)
ans =
0.8147   0.9134   0.2785
0.9058   0.6324   0.5469
0.1270   0.0975   0.9575

>> rand(1,3)
ans =
 0.9649   0.1576   0.9706
```
If you create a very large matrix of random numbers using the `rand` function, the average value will be 0.5.

Notice that we created a 1 by $10^7$ matrix, which required 2 inputs (`rand(1,10e6)`). If we had entered a single value (`rand(10e6)`), the result would have been a $1x10^7$ by $1x10^7$ matrix.
Gaussian Random numbers

◊ `randn(n)`
◊ Also called a normal distribution
◊ Generates numbers with a mean of 0 and a standard deviation of 1
First generate an array of 10 million gaussian random numbers

Use MATLAB to take the mean, and notice that it is very close to 0

Use MATLAB to find the standard deviation, and notice that it is very close to 1
The hist function creates a histogram of the input data.
To generate random numbers between other bounds...

\[ x = (b - a) \cdot r + a \]

- \( a \) and \( b \) are the upper and lower bounds
- \( r \) is the array of random numbers
Although the average is the same for each of these data sets, they have a different standard deviation.
3.7 Complex Numbers

- `complex(x,y)`
- `real(A)` used if A is a complex number
- `imag(A)`
- `isreal(A)`
- `conj(A)`
- `abs(A)`
- `angle(A)`
Script files

◊ Matlab allows commands to be entered interactively at the command prompt, but this is not really appropriate for extended sequences of commands. Instead we can create a Script file.

◊ A script file is simply a text file that contains the sequence of Matlab commands/expressions to follow.

◊ Script files can be created using any text editor or by using the Matlab internal editor.

◊ By convention, script files should be saved with a .m ending. Script files are executed by typing the name of the script without the .m ending. The commands in the script file are executed as if they had been typed in at the command prompt.
Script files

- **Script commands**

  - Run Script

  - Note script can also be run by typing script name at prompt

Script name. Note ends with “.m”

Script output
m-files must be saved in a “file path” that MATLAB knows to look in
An Example

◊ Find the angle where the maximum range occurs
◊ Find the maximum range
◊ **State the problem:** Calculate the range as function of launch angle, velocity and windspeed.

◊ **Describe input and output variables:** gravity \((m s^{-2})\), velocity \((m s^{-1})\), angle \((rad)\), windspeed \((m s^{-1})\). Output distance \((m)\).

◊ **Build a solution:** we can determine the horizontal velocity using trigonometry

\[
(cos(\theta) \ast vel + \text{windspeed}),
\]

and calculate the amount of time the object remains airborne

\[
(2 \sin(\theta) \ast vel)/g.
\]

◊ **Implement a solution:** we can build functions an scripts to realize these models.

◊ **Check the solution:** building plots is a useful way to check your model seems sensible.
A function for distance....

function [ range ] = distance(angle, velocity, windspeed)
%Distance Calculates the range of an object launched at
%a given angle and velocity, with a tailwind of windspeed.
%
%Usage: r= distance(angle,velocity,windspeed)
%Arguments: angle of lanch in radians
% velocity in metres per second
% windspeed in metres per second
%Outputs: distance in metres per second.
%Autor: Tim

gravity = 9.81;
range = (sin(2*angle)*velocity+2*sin(angle)*windspeed)*velocity/gravity;

end
A function for the trajectory....

```matlab
function [Xs, Ys] = trajectory(angle, velocity, windspeed)
    % Distance Calculates the range of an object launched at
    % a given angle and velocity, with a tailwind of windspeed.
    % Usage: r = trajectory(angle, velocity, windspeed)
    % Arguments: angle of launch in radians
    % velocity in metres per second
    % windspeed in metres per second
    % Outputs: distance in metres per second, over a range of up to 1 km.
    % Author: Tim

    gravity = 9.81;
    Xs=[1:1000];
    hvel = cos(angle(1))*velocity+windspeed;
    time = Xs/hvel;
    Ys = time*sin(angle)*velocity - gravity*time.^2./2;
    Ys = (Ys>0).*Ys;
end
```
A script to run everything....

```matlab
%script to graph ranges for an angled launch
theta = [0:0.1:pi/2];
velocity = 100;
windspeed = -30;
polar(theta,distance(theta, velocity, windspeed),'-r');
xlabel('Angle');
ylabel('Range');
hold;

Z = zeros(length(theta),1000);
for ii = 1:length(theta)
    [X,Y] = trajectory(theta(ii),velocity,windspeed);
    Z(ii,:) = Y;
end
plot(X,Z);
xlabel('Distance');
ylabel('Height');
title('Trajectories');
```
Control Flow

◊ So far, the code we have written has simply been a sequence of commands - executed one after the other from top to bottom.

◊ There are many occasions where you want branching or looping of your code. For example, selecting an action depending on user input.

◊ Branching can control the behaviour of your program to depend on the results of computations.

◊ With branching, you can select particular blocks of code to be executed, and consequently, skip over other blocks of code.

◊ Branching depends on *relational and logical operators*...
Flow chart

http://en.wikipedia.org/wiki/Flow_chart
Relational operators

◊ A relational operator takes two numerical operands and yields either a result of True or False.

◊ Matlab uses 1 for True and 0 for False.

◊ Matlab relational operators:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>==</td>
<td>Equal to</td>
</tr>
<tr>
<td>~=</td>
<td>Not equal to</td>
</tr>
<tr>
<td>&gt;</td>
<td>Greater than</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Greater than or equal to</td>
</tr>
<tr>
<td>&lt;</td>
<td>Less than</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Less than or equal to</td>
</tr>
</tbody>
</table>
Examples

◊ Examples:

\[
\begin{align*}
\text{>> 5 > 4} & \quad \% \text{ Result is 1 (True).} \\text{\hspace{1cm} ans =} \\
& 1 \\
\text{>> 3 == 2} & \quad \% \text{ Result is 0 (False).} \\
\text{>> 'a' < 'b'} & \quad \% \text{ Result is 1 (True).} \\
& \% \text{ Characters are ordered by ASCII value.}
\end{align*}
\]

◊ Warning: a common error is to use the assignment operator (=) instead of the relational operator (==).
Caution!

- Avoid using equality and inequality on floating point numbers:

  ```matlab
  >> a = 0;
  >> b = sin(pi);
  >> a == b
  ans = 0                % false!!
  
  >> b
  b = 1.2246e-16
  ```

- Since numbers are represented in the computer by a fixed number of bits, Matlab can only calculate numbers to a certain accuracy or resolution.

- “real” numbers are rarely exact. They have round-off error.
Allow for roundoff error

- If you really need to test for equality between floating point values, allow for roundoff error by doing something like the following:

```plaintext
>> % Check if values a and b are equal (within machine precision).
>> abs(a-b) < 1.0e-14
ans =
   1
```

- The `abs` function calculates the absolute value of a number.

- Reserve the use of the equality and inequality operators for integers only.
Logical Operators

◊ A logical operator takes one or two logical operands and yields a logical result of either 1 (True) or 0 (False).

◊ Matlab logical operators:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>~</td>
<td>Not</td>
</tr>
<tr>
<td>&amp; (&amp;&amp;)</td>
<td>And</td>
</tr>
<tr>
<td></td>
<td>(</td>
</tr>
<tr>
<td>xor</td>
<td>Exclusive Or (Xor)</td>
</tr>
</tbody>
</table>

◊ Note that exclusive or (xor) is performed by the function xor and is not a built in operator.
A truth table defines how these logical operators work:

<table>
<thead>
<tr>
<th>Input</th>
<th>Not</th>
<th>And</th>
<th>Or</th>
<th>Xor</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>y</td>
<td>~x</td>
<td>x &amp; y</td>
<td>x</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Use of Partial Evaluation

◊ Also called “short circuit” evaluation.

- `&&` will evaluate the first expression and immediately return a false value if the first expression is false regardless of the second expression.

- `||` will evaluate the first expression and immediately return a true value if the first expression is true regardless of the second expression.

• Only work for scalars, not matrices.

• Example:

  ```
  >> x = a/b > 10.0
  >> x = (b ~= 0) && (a/b > 10.0)
  ```
Examples

◊ Examples of logical operations:

\[
\text{\texttt{>> \sim 4}} \quad % \text{4 is non-zero, hence 4 is True.} \\
\text{\texttt{\hspace{1em}}\% \text{Not True is False.}}
\]

\[
\text{\texttt{\hspace{1em}ans = \hspace{1em}0}}
\]

\[
\text{\texttt{>> (2 < 3) \& (6 < 8)}} \quad % \text{A True statement.}
\]

\[
\text{\texttt{\hspace{1em}ans = \hspace{1em}1}}
\]
Precedence of relational and logical operators

◊ The precedence of relational operators is lower than that of arithmetic operators.

◊ The precedence of logical operators is lower than that of relational operators.

◊ The rules used when evaluating an expression (according to Chapman Sec. 3.3) are:
  
  • Arithmetic operators are evaluated according to their precedence rules.
  
  • Relational operators are evaluated from left to right.
  
  • All ~ operators are evaluated from left to right.
  
  • All & (and &&) operators are evaluated from left to right.
  
  • All | (and ||) and xor operators are evaluated from left to right.
Example

◊ For example:

    >>> ~ 2 + 3 < 10 + 1

will be evaluated as:

    >>> ~((2 + 3) < (10 + 1))

◊ *Is it?*

◊ Put brackets in when it aids readability (and to avoid bugs!)
Selection

◊ **The if Construct**

◊ The primary branching command is **if-else** construct:

```plaintext
if ControlExpression1
    statement              % Code block 1
    statement
    ...
else
    statement              % Code block 2
    statement
    ...
end
```

◊ The **else** clause is optional.
Example

```matlab
>> x = -1;
>> y = x;
>> if y<0
    y=abs(x)
    end
y = 1
```

◊ The values of the control expressions dictate which blocks of code will be executed.

◊ If the value of a control expression is non-zero, it is considered True and the corresponding block of code will be executed.

◊ If it is considered False and the corresponding block of code will be skipped.

◊ Control expressions typically make use of relational and logical operators.
**elseif**

```plaintext
if ControlExpression1
    statement                % Code block 1
    ...
endif
elseif ControlExpression2
    statement                % Code block 2
    ...
endoelse                         % optional
    statement                % Code block n
endo
```

- Only the *first* expression that evaluates to True “fires”.
Putting it together in an example

Calculate a student grade given a final mark:

```matlab
% Get input from keyboard.
mark = input('Enter the mark: ');

% First check for valid input. Valid marks must be
% between 0 and 100.
if (mark > 100) | (mark < 0)
    disp('Invalid mark');
    return;          % Exit from the program.
end                % Of first if statement
```

continued...
Example (Cont.)

if mark >= 80
    grade = 'HD';
elseif mark >= 70
    grade = 'D';
elseif mark >= 60
    grade = 'CR';
elseif mark >= 50
    grade = 'P';
elseif mark >= 45
    grade = 'N+';
else
    grade = 'N';
end

disp(grade);  % Display the final grade.
Example - explained

◊ The function `input` prints out a message to the user (specified as a character string) and waits for keyboard input.
  • the value entered at the keyboard is then assigned to a variable.
  • in the example above, a variable called `mark` is used to hold the entered value.

◊ An `if..elseif` construct is used to assign a string value to the variable `grade`, depending on the value of `mark`.

◊ The `disp` function is used to print out the value of variables
  • in this case the value of `grade`. 
Nested if statements

```plaintext
if (test 1)
  if (test 2)
    if (test 3)
      end
    end
  end
end
```

• Different to...

```plaintext
if (test 1)
  ...
elseif (test 2)
  if (test 3)
    ...
  end
end
```

• Why?
The switch structure

◊ The switch statement is another form of branching construct.

◊ The switch statement allows you to select a block of code to execute according to the value of a single variable.
Example

switch (str) % The variable str is the variable
% that controls the switching.
% str can be a scalar or a string,
% but not an array.

case {'dog', 'cat', -3} % If str matches one of these items
     disp('A pet or -3'); % then this code block will be
     % evaluated.
% When you have a set of allowable
% values, they must be enclosed in
% curly brackets.

case {1, 2, 3, 4, 5}
     disp('An integer in the range 1 to 5');

case 7
     disp('The value is 7'); % A single case value does not need
% to be put in curly brackets.

otherwise
    disp('Unknown object') % This catches any case not caught
% previously.
% The otherwise case is optional.

end
Good-Practice: Formatting your code

• When formatting your code always indent blocks of code within if statements, typically by 2 (to 4) spaces or one tab character.

• The physical layout of your code should reflect the logical structure of the program.

• A good layout makes the code very much easier to read and debug.

• Be consistent with your spacing.
Repetition (Loops)

◊ Loops allow you to execute a series of statements more than once.

◊ Matlab provides two forms of loop constructs:

1. The *while loop*, which repeatedly executes a block of statements as long as some condition is satisfied.

2. The *for loop*, which repeatedly executes a block of statements a definite number of times.
The **while** loop

- The **while** loop repeatedly executes a block of statements as long as some condition is satisfied.

- The general form of a while loop is:

```
while ControlExpression
    statement       % Code block to execute while
    statement       % in the while loop.
    ...
end              % End of code block.
```

```
statement       % Statement to perform after
                  % the while loop is complete.
```
Execution of the `while` Loop

- If the control expression is True (non-zero), the code block will be executed.
- When the end statement is reached, control returns to the `while` statement.
- If the control expression is still True, the block of code within the `while` loop is re-entered and will be repeated indefinitely until the control expression becomes False (zero).
- Once the control expression becomes False, control is transferred to the first statement after the end statement.
Example – A guessing game

◊ A guessing game in which the user is prompted to guess a pre-defined number.

```matlab
answer = 6;            % Initialise variables.
guess = 1;
guessCount = 0;

while guess ~= answer
    guess = input('Enter your guess:  ');
    guessCount = guessCount + 1;
end

disp('You have guessed correctly.');
```

◊ Warning: make sure the logic that you use for the termination condition is correct – otherwise the loop may never terminate!
The **for** loop

* The **for** loop will execute a block of statements a fixed number of times.

* The general form of the **for** loop is:

```plaintext
for index = expr
    statement  % Code block to execute
    statement  % while in the for loop.
    ...
end        % End of code block.

statement  % Statement to perform after
            % the for loop is complete.
```

* The variable called *index* is the **loop variable** (or **loop index**).

* The control expression *expr* is an **array of index values** over which the block of code will be executed.
The for loop control statement

◊ Almost invariably expr is a sequence of values constructed with the colon operator.

◊ For example, add the numbers from 3 to 16.

```matlab
sum = 0;                 % Initialise a variable to accumulate the sum.
for ii = 3:16
    sum = sum + ii;
end

disp(sum);              % Display the answer.
```

◊ At each successive step of the loop, the loop variable is assigned the next element value from the control expression array.

◊ The number of elements in the control expression array specifies how many times the loop is executed.
Exercise

1. Use a for loop to display the squares of the numbers from 1 to 10.
2. Can you do the same thing without using a loop?
Good programming practice for `for` loops

◊ Never modify the value of the loop variable within the loop - it can make your code very hard to read and understand.

◊ It is almost a tradition among programmers to use a variables `i` and `j` as loop variables. However, in Matlab, `i` and `j` are predefined values representing the square root of `-1`. If you wrote:

```matlab
for i = 3:16
    ...
end
```

you would be overwriting the predefined value of `i`.

◊ Good practice to avoid overwriting built-in values: `ii` and `jj` often used as loop variables.
Example

◊ **Problem:** Find the (approximate) point at which the cosine function crosses the x-axis.

◊ To find where it occurs, we evaluate successive values of the cosine function in increments of 0.01 radians and determine where the sign changes...
The algorithm

- Keep track of the value of the cosine function at the last angle tried.

- Calculate the result of multiplying the cosine at the current angle with the cosine at the last angle.

- If the result is positive, the two values must be on the same side of the x-axis. We need to keep repeating.

- If the result is zero, the new value must be on the x-axis. We can stop repeating - we have found the cross-over point.

- If the result is negative, the signs of the two values must be different. The two points therefore must lie on different sides of the x-axis. We can stop repeating - we have found the cross-over point.
Implementation

lastCosValue = cos(0);
found = false;
theta = 0;

while ~found & theta <= pi % control logic is here
    theta = theta + 0.01;

    currentCosValue = cos(theta);

    if currentCosValue*lastCosValue <= 0
        disp('The cosine function crosses the x axis near the angle');
        disp(theta);
        found = true; % OK to finish
    end % end if

    lastCosValue = currentCosValue; % Current value becomes last value.
end % End for loop.
An implementation of this algorithm:

\[
\text{lastCosValue} = \cos(\pi/4); \quad \% \text{ Create a dummy last value to get started.}
\]

\[
\text{for } \theta = 0 : 0.01 : \pi
\]
\[
\text{currentCosValue} = \cos(\theta); \quad \% \text{ Evaluate the current value.}
\]

\[
\text{if } \text{currentCosValue}*\text{lastCosValue} \leq 0
\]
\[
\text{disp('The cosine function crosses the x axis near the angle');}
\]
\[
\text{disp(}\theta); \quad \% \text{..out of for loop.}
\]
\[
\text{break; } \quad \% \text{end if}
\]

\[
\text{lastCosValue} = \text{currentCosValue; } \quad \% \text{Current value becomes last value.}
\]

\end \% \text{End for loop.}

◊ Remember to make variable names as descriptive as possible

◊ – makes the code easier to read and follow.
A vectorized algorithm:

\[
\theta = [0:0.01:\pi]; \quad \% \text{create an array of angles}
\]

\[
c = \cos(\theta); \quad \% \text{calculate } \cos \text{ of angles}
\]

\[
a = c(1:length(c)-1).\times c(2:length(c)) < 0;
\]

\[
\text{disp('The cosine function crosses the x axis near the angle');}
\]
\[
\text{disp(\theta(a));}
\]

◊ Remember to make variable names as descriptive as possible

◊ – makes the code easier to read and follow.
Nested loops

◊ Loops can be nested.

◊ To access all the elements of a 2D matrix a nested loop is required.

◊ For example, write the code to multiply every element of a matrix by 2:

```matlab
% The size function returns the number of rows
% and columns of a matrix.

[nrows, ncols] = size(m);
for r = 1 : nrows
    for c = 1 : ncols
        m(r, c) = 2*m(r, c);
    end
end
```

◊ Multiplying a matrix by 2 is better done using the single Matlab statement:

```matlab
>> m = m*2;
```

◊ By exploiting Matlab's syntax, we can often avoid writing loops altogether.
Case Study - Cantilever Beam

Diamond Cantilever beam - uniform load

\[ y = -(x^4 - 4x^3 + 6x^2) \]
Solution 1: Loops

```matlab
for ii = 1:101
    X(ii) = (ii-1)/100;
    Y(ii) = - (X(ii)^4 - 4*X(ii)^3 + 6*X(ii)^2);
end
plot(X,Y);
axis([0, 1.2, -5, 1])
title('Deflection of a Cantilever Beam')
xlabel('X')
ylabel('Y')
hold on
plot([0 1],[0 0],'--','color','red')
legend('Uniformly Loaded Beam','Unloaded Beam')
```
Solution 2: Matrices

\[ X = 0:0.01:1; \]
\[ Y = -(X.^4 - 4*X.^3 + 6*X.^2); \]

plot(X,Y);

axis([0, 1.2, -5, 1])
title('Deflection of a Cantilever Beam')
xlabel('X')
ylabel('Y')
hold on
plot([0 1],[0 0],'-','color','red')
legend('Uniformly Loaded Beam','Unloaded Beam')
Problems with scripts

◊ So far, the code we have seen has been in the form of script files.

◊ A script file is simply a set of Matlab statements stored in a file.

◊ The commands in a script file are executed as if they had been typed in at the command prompt.

◊ Problems with script files:
  • Everything has to be in one file.
  • How do you write really big programs that are thousands of lines long?

◊ The general principle - take a problem and break it down into smaller, more manageable tasks.
Problems with Scripts

◊ You are likely to write some bits of code within a script file that you would like to use elsewhere in other programs.

◊ Cutting and pasting bits of code can create variable name clashes and other unexpected side-effects.

◊ All the variables created by the script file are "left behind" when the script finishes.

◊ These variables can cause unexpected problems when subsequent scripts are run.

◊ The variables can accidentally be reused or overwritten by a command typed in at the command prompt or by another script file.

◊ For example, a script that fails to initialise a variable may "inherit" a strange value from a previous script causing unexpected side-effects or errors.
The concept of Workspace (Namespace)

◊ The *workspace* (or *namespace*) can be thought of as an area where variables are stored - a bit like having a blackboard where values of variables are written down for future reference.

◊ The fact that all scripts share the *same workspace* for their variables means that things can get a bit messy.

◊ As each script is run, it leaves behind its "rubbish" (variables) which can then interfere with other scripts.

◊ If you want to ensure nothing interferes with anything else (either between scripts, or within a script) you have to ensure that no variable or function name gets reused.

◊ Clearly this is impossible to achieve in a large program.

◊ A classic problem occurs when creating a variable name that clashes with a function name.
Example

For example, imagine the following code that sums values in a for loop:

```matlab
sum = 0;             % Initialise a variable
                    % called sum.
for ii = 1 : 10
    sum = sum + ii;   % Add the values.
end
```

Imagine later in your code, you attempt to use the built-in Matlab function called sum to add up the values 6 to 20:

```matlab
>> b = sum(6 : 20);
```

Matlab reports back an error:

```matlab
???  Index exceeds matrix dimensions.
```

By initialising a variable called `sum`, we have "shadowed" the function called `sum`.
What happens then?

◊ When your code refers to the name of something,

1. Matlab first looks to see if there is a variable of that name.
2. If found, Matlab will use the data corresponding to that variable.
3. If a matching variable name is not found, Matlab then searches for a script or function file with that name (e.g. sum.m).

◊ Thus, in the previous example, the function sum (in the file sum.m) is hidden behind the "shadow" of the variable called sum.

◊ Matlab's error message reports that it considers the name sum to refer to a 1x1 matrix and that you are attempting to extract elements 6 to 20 from it.
How to resolve the problem

◊ We can of course remove the definition of the variable *sum* with the clear command:

```matlab
>> clear sum   % This clears the variable
    % sum from memory.
```

◊ But this is not convenient if the program uses lots of variables.

◊ To solve (or at least reduce) this problem, we desire a mechanism whereby individual modules of code can operate within their own workspace separate from all other modules of code.

◊ An analogy:

• Imagine a group of people all working simultaneously at a blackboard fighting over who gets to use variable names a, ii, sum, etc. for their calculations.

• Alternatively, imagine the same group of people all working independently using their own piece of paper to do their calculations. Everyone can use a variable called ii on their own piece of paper without interfering with anyone else.
Why Functions

- Break tasks down into manageable units or “building blocks”
- Well specified operation of each building block - inputs and outputs
- Ability to test each building block separately - easier than testing and debugging the “whole”
- Each building block “looks after” its own variable namespace
- Building blocks can be reused (especially “generic” functions) - increased confidence, reduced labour.
- Allow distribution of workload on multiperson projects.
- Allow building blocks to be improved without changing other code.
Matlab Functions

• The mechanism used to code up independent subtasks is that of a Matlab function.
• A function is similar to a script in that it is a set of Matlab commands stored in a file.
• The name of the file is the same as the name of the function with a .m ending.
• A function differs from a script in that it operates within its own private workspace.
• A function is also different in that it can accept arguments and return values, something a script cannot do.
• The typical layout of a .m function file...
How to write a Matlab Function

% First have a contiguous set of comment lines
% to be printed out by the help command.
% These comments should:
%
% 1) Describe what the function does.
% 2) Specify the arguments the function takes.
% 3) Specify what the function returns.
%
% Then have some internal comments that give:
%
% 1) The author of the code.
% 2) The date the code was written.
% 3) Dates of revisions and descriptions of the
% changes.
How to write a Matlab Function

function [output1, output2,...] = fName(input1, input2,...)
    %Your code, which should assign values to output1, output2,...

return    % This is optional.
end

Note: function may have no outputs, eg:
    function drawSquare (sideLength)
and even no inputs
    function insertMyPicture
A Simple Example

Define a function to calculate the area of a triangle. We will call the function `trianglearea`. Consequently, the file holding this function must be called `trianglearea.m`.

```matlab
% TRIANGLEAREA: A function to find the area of a triangle.
% Usage: area = trianglearea(width, height)
% Arguments: width - The width of the triangle.
% height - The height of the triangle.
% Returns: area - The area of the triangle.
% Author: PK
% Date: August 2005

function area = trianglearea(w, h)
    area = w*h/2;
end % function trianglearea
```
Call the function

◊ We can then call this function from the command prompt, or from a script file, or from another function.

```matlab
>> a = trianglearea(3, 4)
a=
   6
```

```matlab
>> b = 10; c = 3;
>> z = trianglearea(b, c)
z=
   15
```
A Close Look

◊ Note that the variable names used within the function are completely independent from the names used when the function is called.

◊ In the statement:

\[
\text{\texttt{z} = trianglearea(b, c)}
\]

the following things occur:

1. The \textit{value} of \( b \) is received in the function \texttt{trianglearea} as a variable called \textit{width}.

2. The \textit{value} of \( c \) is received in the function as a variable called \textit{height}.

3. The variable \textit{area} calculated within the function is returned and assigned to the variable \( z \).
Another Example

% MAXMIN: A function to find the maximum and minimum values in a row vector.
%
% Usage: [maxValue, minValue] = maxmin(v)
%
% Arguments: v - The row vector of elements.
%
% Returns: maxValue - The maximum value in v.
% minValue - The minimum value in v.
%
% Author: PK
% Date: August 2005
function [maxValue, minValue] = maxmin(v)
    [nrows, ncols] = size(v);
    if nrows ~= 1
        disp('This is not a row vector.');
        return;
    end
    if ncols < 1
        disp('The row vector contains no elements');
        return;
    end
maxValue = v(1);
minValue = v(1);

for ii = 2 : ncols
    if v(ii) < minValue
        minValue = v(ii);
    end

    if v(ii) > maxValue
        maxValue = v(ii);
    end

end
end % function maxmin
Anonymous Functions

◊ Normally if you go to the trouble of creating a function, you want to store it for future use.

◊ Anonymous functions are defined inside a script M-file or in the command window, and are only available while they are stored in the workspace window – much like variables.

◊ Suppose you’d like to define a function for natural log called \( \ln \)

◊ \( \ln = @(x) \log(x) \)
  
  • The \( @ \) symbol alerts MATLAB that \( \ln \) is a function
  • The function input is next, inside parentheses
  • Finally the function is defined
The name of the function is called a function handle – in this case it is `ln`.

Notice that function handles are represented with the box symbol in the workspace window.
The name of the function is called a function handle – in this case it is `ln`. Notice that function handles are represented with the box symbol in the workspace window.
Saving Anonymous Functions

- Anonymous functions can be saved as a .mat file – just like anything else listed in the workspace window
- Retrieve anonymous functions using the load command
Function Handles

◊ It is possible to assign a function handle to any user-defined function, including those stored in m-files

◊ If an m-file called distance is stored in the current directory then..

◊ distance_handle = @(t) distance(t)

◊ Function handles make using “function functions” easier

◊ Some functions accept other functions as input

◊ An example is the fplot function described in chapter 5 or the nargin and nargout functions described in this chapter
fplot requires a function in the first input field

Either syntax gives the same result

```
>> fplot('sin(x)', [0,10])
```

```
>> fplot('sin', [0,10])
```

```

```
Function functions can also accept a function handle in place of the function itself – in this case \( \ln \)
Here’s another example where a function handle is assigned to a more complicated expression.

It’s easier to use poly5 in the fplot command, than to type in the whole polynomial.
We can use the function handle again in the fzero function function, which finds the value of x when the function equals zero.
Subfunctions

◊ More complicated functions can be created by grouping functions together into a single file, as subfunctions

◊ Each MATLAB function m-file has one primary function

◊ Subfunctions are added to the primary function, and can have any legitimate MATLAB name
function [add_result, subtract_result] = subfunction_demo(x,y)

% This function both adds and subtracts the elements stored
% in the two input arrays

add_result = add(x,y);
subtract_result = subtract(x,y);

function result = add(x,y)
result = x+y;

function result = subtract(x,y)
result = x-y;

>> x = [1 2 3 4];
>> y = [15 0 4 6];
>> [plus, minus] = subfunction_demo(x,y)

plus = 
   16.00    2.00    7.00   10.00

minus = 
  -14.00    2.00   -1.00   -2.00
Need to revisit how to solve a problem as it is the same in planning an algorithm