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

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>

http://en.wikipedia.org/wiki/Flow_chart
Examples

• Examples:

  >> 5 > 4                % Result is 1 (True).
  ans =
   1

  >> 3 == 2               % Result is 0 (False).
  >> 'a' < 'b'            % Result is 1 (True).
  % Characters are
  % ordered by ASCII value.

• 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:

  >> 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:

  >> % 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:

  Operator      Meaning
  ~                Not
  & (&&)           And
  | (||)           Or
  xor             Exclusive Or (Xor)

• Note that exclusive or (xor) is performed by the
  function xor and is not a built in operator.
Truth table

• 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>F</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td></td>
<td>T</td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td></td>
<td>T</td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td></td>
<td>T</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:

  ```
  >> ~4       % 4 is non-zero, hence 4 is True.
  ans =
   0
  
  >> (2 < 3) & (6 < 8)   % A True statement.
  ans =
   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:
  1. Arithmetic operators are evaluated according to their precedence rules.
  2. Relational operators are evaluated from left to right.
  3. All `~` operators are evaluated from left to right.
  4. All `&` (and `&&`) operators are evaluated from left to right.
  5. 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:

    if ControlExpression1
      statement                  % Code block 1
      statement
      ...
      else
      statement
      statement
      ...
    end

• The else clause is optional.

elseif

• Only the first expression that evaluates to True “fires”.

Example

>> x = -1;
>> y = x;
>> if y<0
    y = abs(x)
else
  y = 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 the control expression evaluates to 0, it is considered False and the corresponding block of code will be skipped.

• Control expressions typically make use of relational and logical operators.
Putting it together in an example

- Calculate a student grade given a final mark:

```
% 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

% Mark is in range 80 - 100.
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

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

    % Different to...
    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

```matlab
switch (str)  
    case {'dog', 'cat', -3}  
        disp('A pet or -3');  
    case {1, 2, 3, 4, 5}  
        disp('An integer in the range 1 to 5');  
    case 7  
        disp('The value is 7');  
    otherwise  
        disp('Unknown object');  
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.

## Defensive programming

- It is strongly suggested that when you have a set of conditional statements you should include tests for "the dumb cases that will never happen".
- When the dumb cases do happen, your code will detect them and print out an error message.
- The extra minute spent putting these additional tests in may save you many hours of debugging.
- Testing for these cases typically covers testing the necessary pre-conditions for your code to work correctly.
- You should `assert` that pre-conditions hold. If an assertion fails, exit with an error message.
  - example: see the marks to grades program
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:

```
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.

**Exercise**

- **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:

```
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

```matlab
theta = 0;
currentCosValue = cos(theta);
found = false;
while ~found & theta <= pi % control logic is here
    theta = theta + 0.01;
    lastCosValue = currentCosValue; % Store last value
    currentCosValue = cos(theta); % Get new value
    if currentCosValue*lastCosValue <= 0
        disp('The cosine function crosses the x axis near the angle');
        disp(theta-0.005);
        found = true; % OK to finish
    end
end
% End while loop
```

Alternative Implementation

```matlab
theta = 0;
currentCosValue = cos(theta);
found = false;
while ~found && theta <= pi % control logic is here
    theta = theta + 0.01;
    lastCosValue = currentCosValue; % Store last value
    currentCosValue = cos(theta); % Get new value
    found = currentCosValue*lastCosValue <= 0;
    if found
        disp('The cosine function crosses the x axis near the angle');
        disp(theta-0.005);
    end
end
% End if
% End while loop
```
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:
  >> m = m*2;
  
  % By exploiting Matlab's syntax, we can often avoid writing loops altogether.
  
Case Study - Cantilever Beam

Cantilever beam - uniform load

Image: [Link to image]

\[ y = -(x^4 - 4x^3 + 6x^2) \]

Hahn & Valentine, Section 14.1

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

```matlab
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')
```