# Programming in MATLAB

Trevor Spiteri
trevor.spiteri@um.edu.mt
http://staff.um.edu.mt/trevor.spiteri

Department of Communications and Computer Engineering
Faculty of Information and Communication Technology
University of Malta

17 February, 2008

# Outline

# Outline

# Relational operators

Table : Relational operators

| | |
|---|---|
| `a == b` | Equal to. |
| `a ~= b` | Not equal to. |
| `a < b` | Less than. |
| `a > b` | Greater than. |
| `a <= b` | Less than or equal to. |
| `a >= b` | Greater than or equal to. |

- These operators can operate on single values or on arrays.
- The result is of type `logical`.
- The `logical` type can be `0` or `1` only.

# Logical operators

Table : Logical operators

| | |
|---|---|
| `~a` | NOT `a`. `1` if `a` is zero. |
| `a & b` | `a` AND `b`. `1` if both `a` and `b` are non-zero. |
| `a \| b` | `a` OR `b`. `1` if either `a` or `b`, or both, is non-zero. |
| `xor(a, b)` | `a` XOR `b`. `1` if `a` or `b` are non-zero, but not both. |

- These four logical operations can be performed on single values or on arrays.
- The result is of type `logical`.

# Short-circuit operators

Table : Short-circuit operators

| | |
|---|---|
| `a && b` | 1 if both `a` and `b` are non-zero. |
| `a || b` | 1 if either `a` or `b`, or both, are non-zero. |

- The short-circuit operators operate on scalar expressions only.
- The second expression is evaluated only if it needed for the result.
- For the AND operation `a && b`, if `a` is zero, the result will be `0` whatever `b` is. In this case, `b` is not evaluated.
- For the OR operation `a || b`, if `a` is non-zero, the result will be `1` whatever `b` is. In this case, `b` is not evaluated.

# Accessing arrays using logical arrays

- Logical arrays can be used to access values of an array.
- Let `a` $= \begin{bmatrix} 2 & 2 \\ 2 & 2 \end{bmatrix}$ and `b` $= \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$.
- Suppose we write: `>> a_ge_b = a >= b`
- Now `a_ge_b` is the logical array $\begin{bmatrix} 1 & 1 \\ 0 & 0 \end{bmatrix}$.
- `b(a_ge_b)` refers to the values of `b` for which our expression is true.
- `b(a_ge_b)` has the values 1 and 2.
- To add 10 to these numbers, we type:
  `>> b(a_ge_b) = b(a_ge_b) + 10`
- Now, `b` $= \begin{bmatrix} 11 & 12 \\ 3 & 4 \end{bmatrix}$.
- This works only with logical arrays, and does not work with a normal array containing ones and zeros.

# The `find` function

- The `find` function evaluates a logical expression and returns the indices of the true values.
- Let `a` $= \begin{bmatrix} 1 & 2 & 0 & 0 \\ 3 & 0 & 0 & 10 \end{bmatrix}$.
- To find the true values, we type: `>> f = find(a)`
- `f` is a vector containing the linear indices 1, 2, 3 and 8.
- To obtain the true values, we type: `>> t = a(f)`
- Now `t` is a vector containing the values 1, 3, 2, and 10.
- There are other calling methods for `find`. To read about them, type: `>> help find`

# Outline

# The `if` statement

- The most-used conditional statement is the `if` statement:

  `if` *expression*$_1$

  *statements*$_1$

  `elseif` *expression*$_2$

  *statements*$_2$

  `else`

  *statements*$_3$

  `end`
- The statements following the first true expression are executed.
- The `elseif` and `else` parts are optional.
- `if` statements must always finish with `end`.
- If required, more than one `elseif` part can be used.
- `if` statements can be nested.

# The `switch` statement

- The `switch` statement selects among several cases.

  `switch` *switch expression*
      `case` *case expression*
          *statements*
      `case {`*case expression*$_1$`,` *case expression*$_2$`,` ...`}`
          *statements*
      `otherwise`
          *statements*
  `end`
- The switch expression can be a scalar or a string.
- The statements following the first `case` where the case expression matches the switch expression are executed.
- Several cases can be grouped inside curly brackets.
- If no match is found, the `otherwise` statements are executed.

# When to use loops

- It is sometimes easy to perform operations on many values at once without explicit loops.
- When it is possible to use array operations, explicit loops should be avoided.
- There are two kinds of loop statements.
- The `for` loop repeats statements for a specific number of times.
- The `while` loop repeats statements for an indefinite number of times.

# The `for` loop

- The `for` loop repeats statements for a number of times.

  `for` *variable* `=` *expression*
  *statements*
  `end`
- The columns of *expression* are stored in *variable* one at a time.
- For each column, the statements in the loop are executed.
- The most common expression is of the form `j:k`.

  ```
  for f = 1:10
  ```
- In this case, the statements are repeated for 10 times.
- In the first iteration `f` is 1, in the last iteration `f` is 10.
- The `break` statement can be used to exit the `for` loop.
- The `continue` statement passes control to the next iteration.
- `for` loops can be nested.

# The `while` loop

- The `while` loop repeats statements for an indefinite number of times.

  `while` *expression*
      *statements*
  `end`
- The statements are repeated while the expression is true.
- The `break` statement exits the `while` loop.
- The `continue` statement passes control to the next iteration.
- `while` loops can be nested.

# Outline

# Numbers

Table : Rounding numbers

| | |
|---|---|
| `floor(x)` | Rounds $x$ to the nearest integer towards $-\infty$. |
| `round(x)` | Rounds $x$ to the nearest integer. |
| `ceil(x)` | Rounds $x$ to the nearest integer towards $\infty$. |
| `fix(x)` | Rounds $x$ to the nearest integer towards 0. |

Table : Exponential and logarithmic functions

| | |
|---|---|
| `exp(x)` | Exponential, $e^x$. |
| `sqrt(x)` | Square root, $\sqrt{x}$. |
| `log(x)` | Natural logarihtm, $\ln x$. |
| `log10(x)` | Common (base 10) logarithm, $\log_{10} x$. |

# Complex numbers and sign

Table : Complex numbers and sign

| | |
|---|---|
| `abs(x)` | The absolute value, $\lvert x \rvert$. |
| `angle(x)` | The phase angle in radians, $\arg(x)$. |
| `sign(x)` | The signum function: |
| | 0 if $x = 0$; |
| | $+1$ if $x$ is real, $x > 0$; |
| | $-1$ if $x$ is real, $x < 0$; |
| | `x ./ abs(x)` for non-zero, complex $x$. |
| `conj(x)` | The complex conjugate, $x^*$. |
| `real(x)` | The real part, $\Re\{x\}$. |
| `imag(x)` | The imaginary part, $\Im\{x\}$. |

# Trigonometric functions

Table : Trignometric functions

| | |
|---|---|
| `sin(x)` | The sine, $\sin x$. |
| `cos(x)` | The cosine, $\cos x$. |
| `tan(x)` | The tangent, $\tan x$. |
| `csc(x)` | The cosecant, $\csc x$. |
| `sec(x)` | The secant, $\sec x$. |
| `cot(x)` | The cotangent, $\tan x$. |

- These functions work in radians. There are corresponding functions which work in degrees: `sind`, `cosd`, `tand`, `cscd`, `secd`, and `cotd`.
- There are hyperbolic versions of these functions: `sinh`, `cosh`, `tanh`, `csch`, `sech`, and `coth`.

# Inverse trigonometric functions

Table : Inverse trignometric functions

| | |
|---|---|
| `asin(x)` | The inverse sine, $\sin^{-1} x$. |
| `acos(x)` | The inverse cosine, $\cos^{-1} x$. |
| `atan(x)` | The inverse tangent, $\tan^{-1} x$. |
| `acsc(x)` | The inverse cosecant, $\csc^{-1} x$. |
| `asec(x)` | The inverse secant, $\sec^{-1} x$. |
| `acot(x)` | The inverse cotangent, $\tan^{-1} x$. |

- These functions work in radians. There are corresponding functions which work in degrees: `asind`, `acosd`, `atand`, `acscd`, `asecd`, and `acotd`.
- There are hyperbolic versions of these functions: `asinh`, `acosh`, `atanh`, `acsch`, `asech`, and `acoth`.

# Script files

- A script file is an external file with a sequence of statements.
- Script files have a `.m` extension.
- Typing the filename will execute all statements in the script as if they were written in the command window.
- When running a script, the value of each statement is displayed in the command window.
- Writing the `;` character after a statement will suppress the output.
- The `;` character can separate different statements on one line.
- The `,` character separates staments without suppressing the output.
- The `echo on` command turns on echoing of commands.
- A long line can be split using three dots (`...`).

# Defining functions

- To write a function, create an external `.m` file.
- The name of the file must be the name of the function.
- The first line must contain the syntax definition of the function.
- For example, function `foo` can be defined in the file `foo.m`:

*foo.m*

```
1 function [square, cube] = foo(x)
2 %FOO Calculates square and cube.
3 square = x .^ 2;
4 cube = x .^ 3;
```

- The parameter `x` is in parenthesis in the function definition.
- The return values `square` and `cube` are in square brackets.
- Line 2 contains a description of the function.
- Note that `.^` is used, allowing for an input vector or matrix.

# Function parameters

- Function parameters are passed by value.
- Consider the function `multiply` below.

*multiply.m*

```
function ret = multiply(x, multiplier)
x = x * multiplier;
ret = x;
```

- The value of `x` changes inside the function only.
- Suppose we type:

```
>> num = 5
>> num2 = multiply(num, 2)
```

- The value of `num` is not changed by the function call.
- So `num` = 5 and `num2` = 10.

# Local and global variables

- Variables used inside functions are local to that function only.
- Once the function exits, all local variables are erased.
- To use the workspace variables, we use global variables.
- Global variables can be declared using the `global` keyword.

*vars.m*

```
function [sum1, sum2] = add(num1, num2)
l = 5;
sum1 = num1 + l;
global g;
g = g + num2;
sum2 = g;
```

- In this function, `l` is local to the function only.
- g is a global variable, and is defined in the workspace.
- The workspace variable g changes after a call to this function.

# Subfunctions

- A subfunction is a function that is only visible to other functions in the same file.
- We can rewrite our previous `foo` function as follows:

*foo.m*

```
1 function [square, cube] = foo(x)
2 %FOO Calculates square and cube.
3 square = pow(x, 2);
4 cube = pow(x, 3);
5 %-------------------------
6 function ret = pow(x, n)
7 %POW subfunction
8 ret = x .^ n;
```

- In line 6, note that the square brackets may be omitted when there is only one return value.
- Also note that the parameter name (`x`) can be reused in the subfunction.

# Nested functions

- Functions may be written within other functions.
- A nested function is different from a subfunction.
- A nested function is always terminated with the `end` keyword.
- When one function in a `.m` file is terminated with `end`, all functions in the file must be terminated with `end`.
- While subfunctions cannot access variables local to the primary function, nested functions can access all variables of all of their outer functions.

# Example of nested functions

- Suppose we have the function `a`.

*a.m*

```
function a(x)
    function a_1(x) % nested in a
        function a_1_i(x) % nested in a_1
        end % function a_1_i
    end % function a_1
    function a_2(x) % nested in a
    end % function a_2
end % function a
```

- Functions can be called from the level immediately above. `a` can call `a_1` and `a_2`. `a_1` can call `a_1_i`.
- Functions may be called from the same level. `a_1` can call `a_2`. `a_2` can call `a_1`.
- Functions may be called from any lower level. `a_1` can call `a`. `a_1_i` can call `a_1` and `a`. `a_2` can call `a`.

# The number of input arguments

- Sometimes a function acts in different ways depending on how many inputs it has.
- Suppose we want a function to find the area of a rectangle from its two sides.
- For a square, only one side is required.
- The `nargin` function returns the number of inputs.
- The function `calc_area` can be written as:

*calc_area.m*

```
function ret = calc_area(x, y)
if (nargin == 1)
    ret = x .* x;
elseif (nargin == 2)
    ret = x .* y;
end
```

# The number of output arguments

- The `nargout` function is used to determine the number of output arguments.
- Suppose we need to write a function to give us the circumference of a circle.
- If two outputs are required, the second output is to be set to the area.
- The function `circ_info` can be written as follows.

*circ_info.m*

```
function [circumference, area] = circ_info(radius)
circumference = 2 * pi * radius;
if (nargout == 2)
    area = pi * radius .^ 2;
end
```

# Outline

# Data files

- Typically, ASCII data files have some lines of text at the beginning.
- These lines are called the header.
- After the header, there are lines of data arranged in rows and columns
- The numbers in a row might be separated by spaces or commas.
- MATLAB provides several ways to import ASCII and binary data.
- There are commands that import the data.
- There is also the Import Wizard that leads you through the import process.

# Importing data

- The Import Wizard makes it easy to import both ASCII and binary data.
- Sometimes it can be useful to load data using commands in a script.
- The `load` command is used to load ASCII data from a file. The file header must be removed beforehand for this to work.
- The command `load filename.dat` creates a matrix `filename` with the data.
- The command `var = load('filename.dat')` creates a matrix `var` with the data.
- The command `var = xlsread('filename')` imports the Microsoft Excel file `filename.xls` into the array `var`.
- Other commands used to import data include `csvread`, `dlmread`, `textread`, and `importdata`.

# Exporting data

- The `dlmwrite` command can be used to export an array in ASCII format.
- Suppose that `var` $= \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \end{bmatrix}$.
- We want to save this matrix to the file `filename.out`.
- We want numbers in a row to be separated by a `;` character.
- We type: `>> dlmwrite('filename.out', var, ';')`
- This creates the file `filename.out`.

*filename.out*

```
1;2;3;4
5;6;7;8
```

- Other commands used to export data include `save`, `csvwrite`, and `xlswrite`.