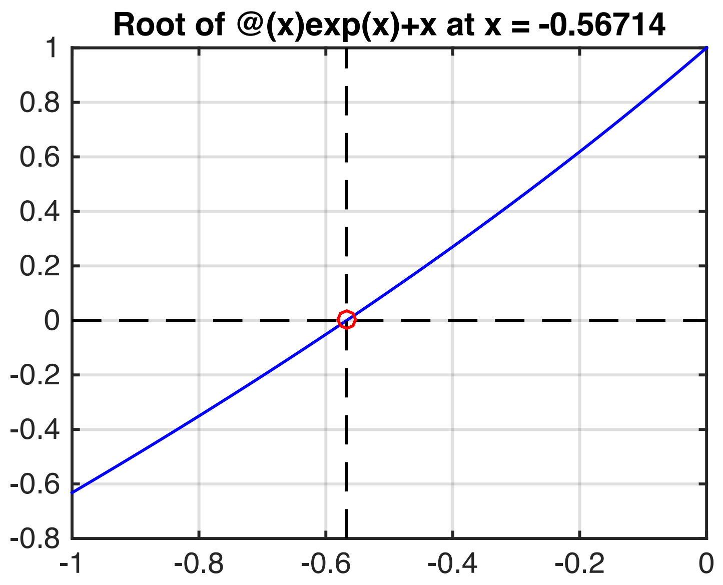
UBC MECH 221: MATLAB Tutorial 2

Logic, Loops, et cetera



```
>> f = @(x) exp(x) + x;
>> bisection(f,[-1,0],0.0001)
```

Patrick Walls
October 18, 2016

Contents

1 Introduction

MATLAB is the *matrix laboratory* and this tutorial is a tour of logic, loops and an assorted collection of helpful tools (et cetera). The best way to learn from these notes is to open a new MATLAB workspace, enter the commands presented, observe the output, try variations of each command, function and script, and refer to the documentation as needed.

Learning Goals

1. **Logic**: relational operators, logical operators, `if` statements
2. **Loops**: `for` loops and `while` loops
3. **et cetera**: `tic toc`, `disp`, `Ctrl+C`, function handles

Instructions

1. Find a computer with MATLAB or Octave installed (or use Octave online as described below).
2. Commands entered in the command window are presented with a grey background:

   ```
   >> x = linspace(-pi,pi,1000); y = sin(x); plot(x,y);
   ```

   where the symbol `>>` represents the MATLAB prompt, and m-files are presented as:

   ```
   function y = fun(x)
   % Compute the square of x
   y = x*x
   end
   ```

3. Enter each command listed in the sections below. It is important to type these commands for yourself and to not copy and paste the text. You must memorize the basic commands and functions.
4. Understand the output and try variations of each command to test your understanding.

Resources

1. MATLAB is now freely available to download for registered UBC students:

   `students.engineering.ubc.ca/success/software`

2. Visit `mathworks.com/help/matlab` for more documentation.
3. Octave is a free, open-source MATLAB clone and is sufficient for this assignment. Octave is available to download at `gnu.org/software/octave`.
4. You can also use Octave online at `octave-online.net`.

2 Logic

2.1 Relational Operators

MATLAB has relational operators to make comparisons between scalars or arrays (of the same size), and comparisons return a `logical` data type: `1` (True) and `0` (False).

`<`	less than
`<=`	less than or equal to
`>`	greater than
`>=`	greater than or equal to
`==`	equal
`~=`	not equal

Check out the documentation:

`http://www.mathworks.com/help/matlab/ref/relationaloperators.html`

Create two arrays of random integers and compare them element by element:

```
>> A = randi([0,10], 5), B = randi([0,10], 5)
>> A < B, A > B, A == B
```

Create two arrays of random integers and compare them element by element:

```
>> A = randi([0,10], 5), B = randi([0,10], 5)
>> A <= B, A >= B, A ~= B
```

Access elements of a matrix using a logical array. For example, create two random matrices and return the values where the arrays are equal:

```
>> A = randi([0,5], 3), B = randi([0,5], 3)
>> A(A == B)
```

2.2 Logical Operators

MATLAB has logical operators to combine relational operators:

`&`	and
`|`	or
`~`	not

Check out the documentation:

http://www.mathworks.com/help/matlab/logical-operations.html

Create random matrices `A` and `B` and find the elements where `A` is positive and larger than the corresponding element of `B`:

```
>> A = randi([-10,10], 4), B = randi([-10,10], 4)
>> ( A > 0 ) & ( A > B )
```

Create a random matrix `A` and find the elements of `A` that are strictly between -3 and 3 but not zero:

```
>> A = randi([-6,6], 4)
>> ( A < 3 ) & ( A > -3 ) & ( A ~= 0 )
```

Equivalently, we could write:

```
>> A = randi([-6,6], 4)
>> ( abs(A) < 3 ) & ( A ~= 0 )
```

Create a random matrix and find the elements equal to 1 or -2:

```
>> A = randi([-5,5], 4)
>> ( A == 2 ) | ( A == -1 )
```

2.3 `if` Statements

Writing computer programs relies on executing certain blocks of code depending on the state of the program: if (*this*) then (*do that*) otherwise (*do this*). The general form of an `if` statement in MATLAB uses the keywords `if`, `elseif`, `else` and `end` (however `elseif` and `else` are optional). Check out the documentation:

http://www.mathworks.com/help/matlab/ref/if.html

Write a MATLAB script to determine if the roots of a quadratic polynomial $ax^2 + bx + c$ are real and distinct, real and repeated or complex by computing the discriminant $b^2 - 4ac$ (and notice that we use the `disp` function to display strings (ie. text)):

```
% Coefficients of the polynomial ax^2 + bx + c
a = 2; b = -3; c = 5;
discriminant = b^2 - 4*a*c;

if discriminant > 0
    disp('Roots are real and distinct')
elseif discriminant < 0
    disp('Roots are complex')
else
```

```
    disp('Roots are real and repeated')
end
```

Create a random matrix of zeros and ones and and determine if the matrix is invertible:

```
A = randi([0,1],3)
if det(A) == 0
    disp('A is not invertible!')
else
    disp('A is invertible!')
end
```

Choose a random number from a normal distribution and see if it is greater than zero (and notice that we are using the `num2str` function (which converts a numeric type to a string) and the string concatenation operator `[ ]`):

```
x = randn;
if x > 0
    disp([num2str(x),'is greater than zero!'])
else
    disp([num2str(x),'is less than or equal to zero!'])
end
```

Take the sum of two random numbers between 1 and 6 as if rolling dice:

```
roll = randi([1,6],1,2)
total = sum(roll)
if total == 2
    disp('Snake Eyes!');
elseif total == 7
    disp('Lucky 7!');
elseif (roll(1) == roll(2))
    disp('A pair!')
else
    disp('Roll again.')
end
```

Write a function called `max_norm` with input parameters `x` and `y` and output `z` equal to the array with the larger norm:

```
function z = max_norm(x,y)
% Return the vector with larger norm (and return x if both have equal norms).
if norm(y) > norm(x)
    z = y;
else
    z = x;
end
```

```
end
```

Call the function with various input:

```
>> max_norm([0,0],[1,1])
>> max_norm([0,0,2],[1,1,1])
```

3 Loops

A loop is a method to execute a block of code repeatedly. There are two types of loops:

`for` loop: *repeat code block a designated number of times*

`while` loop: *repeat code block until a terminating condition is met*

Be careful when writing loops or else you may inadvertently create an infinite loop! But always remember that you can stop a loop (or any MATLAB process) by hitting `Ctrl+C`. Check out the documentation:

`http://www.mathworks.com/help/matlab/matlab_prog/loop-control-statements.html`

3.1 `for` Loops

The general form of a `for` loop uses the keywords `for` and `end` and designates a loop variable to iterate over an array. For example, the following MATLAB script sets `i` as the loop variable and the code block is executed once for each value in the array:

```
for i = 1:10
    disp(['Current iteration in the loop:  ',num2str(i)])
end
```

Display the squares up to 25:

```
for i = 1:5
    squared = i^2;
    disp([num2str(i),'squared is equal to ',num2str(squared)])
end
```

Loop over the columns of the identity matrix:

```
for column = eye(3)
    disp(column)
end
```

Loop over the columns of a matrix:

```
A = [1:5; 6:10]
for column = A
    disp(column)
end
```

In MATLAB's language, a string is also an array – an array of characters. For example, write a function with one input paramter `name` which displays the characters in `name` one at a time:

```
function cheer(name)
% Display the letters of name
% Input:
%   name: any string

for letter = name
    % Skip empty spaces
    if letter ~= ''
        disp(['Give me a ', letter, '!']);
    end
end

disp('What does that spell?!');
disp([name,'!']);

end
```

Call the function:

```
>> cheer('Math')
>> cheer('Mech 2')
```

3.2 `while` Loops

The general form of a `while` loop uses the keywords `while` and `end` with a logical expression which terminates the loop when false. **Be careful** when writing `while` loops and keep in mind that you can stop a process by `Ctrl+C`. For example:

```
n = 10;
while n >= 0
    disp(n)
    n = n - 1;
end
disp('Blast off!')
```

Write a while loop which randomly picks 0 or 1 and stops when 0 is picked:

```
while (randi([0,1]) == 1)
    % Display the result when 0 is generated
    disp('0')
end
disp('1')
```

3.3 Example: Constructing recursive sequences of known length

The Fibonacci sequence is 1, 1, 2, 3, 5, 8, 13, . . . where the next number is the sum of the previous two numbers in the sequence. Check out:

`https://en.wikipedia.org/wiki/Fibonacci_number`

Write a function which computes the first N Fibonacci numbers.

```
function seq = fibonacci(N)
% Compute the first N Fibonacci numbers.

% Initialize the sequence with the first 2 Fibonacci numbers
seq = [1,1];

for i = 3:N
    seq(i) = seq(i-1) + seq(i-2);
end
```

```
>> fibonacci(10)
```

3.4 Example: Constructing recursive sequences of unknown length

A Collatz sequence is a sequence $c_1, c_2, c_3, \ldots$ starting with a positive integer c_0 and is defined by:

$$c_{n+1} = \begin{cases} \dfrac{c_n}{2} & \text{if } c_n \text{ is even} \\ 3c_n + 1 & \text{if } c_n \text{ is odd} \end{cases}$$

The Collatz conjecture states that a Collatz sequence always reaches 1 for any initial value c_1. For example, let $c_1 = 10$ then the sequence

$$\begin{aligned} c_1 &= 10 \\ c_2 &= 5 \\ c_3 &= 16 \\ c_4 &= 8 \\ c_5 &= 4 \\ c_6 &= 2 \\ c_7 &= 1 \end{aligned}$$

Check out:

https://en.wikipedia.org/wiki/Collatz_conjecture

Write a function called `collatz` with one input parameter `c1` and returns the Collatz sequence starting at `c1` as an array `C`. Notice that we use the MATLAB function `rem` which computes the remainder after division (see `https://www.mathworks.com/help/matlab/ref/rem.html`).

```
function C = collatz(c1)
% Construct the Collatz sequence starting at the positive integer c1.

if rem(c1,1) ~= 0
    disp('Initial value is not an integer.')
    C = [];
    return
elseif c1 <= 0
    disp('Initial value is not positive.')
    C = [];
    return
end

% Initialize the array to hold the Collatz sequence
C = [c1];
% Initialize the index
n = 1;
% Compute the sequence until the value 1 is reached
while C(n) ~= 1
    if rem(C(n),2) == 0
        C(n+1) = C(n)/2;
    else
        C(n+1) = 3*C(n)+1;
    end
    n=n+1; % Update the index
end
end
```

3.5 Example: Finding roots of equations

The bisection method is a simple way to find roots of equations. The idea is that if $f(x)$ is a continuous function on a closed interval $[a, b]$ such that $f(a)f(b) < 0$ (the values have opposite sign) then there must be some point $c \in [a, b]$ such that $f(c) = 0$ by the Intermediate Value Theorem. The bisection method computes $f(x)$ at the midpoint $(a + b)/2$ and then chooses the subinterval $[a, (a+b)/2]$ or $[(a+b)/2, b]$ such that $f(x)$ changes sign over that subinterval and then repeats the process. Check out:

https://en.wikipedia.org/wiki/Bisection_method

```
function [root, total_time, iterations] = bisection(f, interval, tolerance)
% Find an approximate root of a function in a given interval
% using the bisection method
% Input:
%   f: function handle, a vectorized function for which we seek a root c
%   interval: vector of length 2, the interval containing a root
%   tolerance: number, the algorithms stops when |f(c)| < tolerance
% Output:
%   root: number satisfying |f(c)| < tolerance
%   total_time: time taken for the method to find root
%   iterations: number of iterations of the method required to find root

tic; % Start the timer
a = interval(1); b = interval(2); % Find a root in the set interval [a,b]
error = max(abs([f(a),f(b)])); % Initialize the error
c = a; % Set the value of c, in case error <= tolerance at the start
iter = 0; % Initialize the number of iterations

% Check the end points
fa = f(a); fb = f(b);
if (fa*fb > 0)
    disp('Cannot guarantee a solution in the given interval.')
    root = NaN; iterations = 0; total_time = 0;
    return % Stop the process
elseif fa == 0
    root = a; iterations = iter; total_time = toc;
    return
elseif fb == 0
    root = b; iterations = iter; total_time = toc;
    return
end

while (error > tolerance)
    iter = iter + 1; % Count the iterations
    c = (a+b)/2; % Choose the midpoint of the interval
    fa = f(a); fc = f(c); fb = f(b);
        if fc == 0
            % Found the root exactly - end the while loop
            root = c;
            iterations = iter;
            total_time = toc;
            break
        elseif (fa*fc < 0)
            % Change of sign of f from a to midpoint
            % The new interval is [a,c]
            b = c;
        elseif (fb*fc < 0)
            % Change of sign of f from midpoint to b
            % The new interval is [c,b]
            a = c;
        end
    error = abs(fc); % The new error is the value |f(c)|
end

root = c; % An approximate root
iterations = iter; % The number of iterations to obtain the root
total_time = toc; % Record the time elapsed in the computation

% Plot the function on the interval [a,b]
x = linspace(interval(1),interval(2),100); y = f(x);
plot(x,y,'b');
title(['Root of ', func2str(f),'at x = ', num2str(c,5)]);
hold on; grid on;

% Plot lines indicating the position of the root
yl = ylim; xl = xlim;
plot([c,c],[yl(1),yl(2)],'k--',[xl(1),xl(2)],[0,0],'k--',c,0,'ro');
hold off;

disp(['Found solution x = ', num2str(root,20), ...
    ' after ', num2str(iterations), ' iterations in ', ...
    num2str(total_time),' seconds.']);
end
```

4 et cetera

The following is a list of useful MATLAB commands that we will use often.

4.1 `disp` and `num2str` and string concatenation [ ]

Use the `disp` command to display text in the command window, `num2str` to convert numbers to text and `[  ]` operator to concatenate text:

```
>> disp(['The value of pi up to 30 significant digits:  ', num2str(pi,30)])
```

4.2 `Ctrl+C`

To end a process (like an infinite loop or a long computation), hit `Ctrl+C`.

4.3 `tic toc`

Record the time for a program to run by using `tic toc`:

```
tic; % Start the timer
for i = 1:100
    disp(['Loop number ', num2str(i)])
end
% Display the elapsed time
stop_time = toc;
disp(['100 loops in ', num2str(stop_time), ' seconds.'])
```

4.4 Function Handles

We can define functions in the command line using function handles:

```
>> f = @(x) x.^2 + 2*x + 1;
>> f(0), f(2), f(-1)
```

Function handles can be passed in as inputs into other functions. Check out the documentation:

`http://www.mathworks.com/help/matlab/function-handles.html`

Write a function which takes a function handle and interval as input and then plots the corresponding figure:

```
function my_plot(f,interval)
% Plot the function f in the interval.
% Input:
%   f: function handle defining a vectorized function
%   interval: vector of length 2, the interval of the plot

x = linspace(interval(1),interval(2),100);
y = f(x);
plot(x,y);
title(['A plot of the function ',func2str(f)])

end
```

Execute this function with different function handles as inputs:

```
>> f = @(x) sin(2*pi*x);
>> my_plot(f,[0,1])
>> f = @(x) exp(-0.1*x.^2) .* cos(10*x);
>> my_plot(f,[-3,3])
```