FMO6 — Web: https://tinyurl.com/ycaloqk6 Polls: https://pollev.com/johnarmstron561

Lecture 2

Dr John Armstrong

King's College London

August 22, 2020

Recap

- Last week we learned how to use MATLAB as a calculator
- We mastered the difference between `*` and `.*`
- We saw an example of a sophisticated calculation: calculating the cumulative distribution function of the normal distribution using the rectangle rule.
- We saw that this code was too hard to follow and too long to type
- We started looking at *functions* which help solve this problem.

A simple function

```
function [ x, y ] = polarToCartesian( r, theta )
x = r * cos( theta );
y = r * sin( theta );
end
```

A complex function

```
function [ result ] = cumulativeNormal( x )
%CUMULATIVENORMAL computes c.d.f of normal distribution
a = 0;
b = 1;
N = 1000;
h = (b-a)/N;
s =  a + ((1:N) - 0.5) * h;
fValues = s.^(-2) .* exp( -(( x + 1 - 1./s).^2)/2 );
integral = h * sum( fValues );
result = 1 / sqrt( 2 * pi ) * integral;
end
```

Complex to write, but easy to use.

Tests

> It it isn't tested, it doesn't work.

How might you test the cumulative normal function?

```
function testCumulativeNormal ()
x = 0.3;
assert(cumulativeNormal(x) > 0.0 );
assert(cumulativeNormal(x) < 1.0 );
assert(abs( cumulativeNormal( -20.0) ) ...
     < 0.001 );
assert(abs( cumulativeNormal(-x) + ...
     cumulativeNormal(x) - 1 ) <0.001);
assert(abs(cumulativeNormal( 2.0 ) - 0.975)...
     < 0.01 );
end
```

Unit tests

- A Unit test is a function which tests your code.
- Unit tests should be completely automated. You don't read the output, you assert that is correct.
- Tests that pass don't print anything at all. Only tests that fail should print errors.
- You can't test real numbers for equality, you should use inequalities.
- You can right one big test that runs all your tests. There are also "unit testing frameworks" to automate things and print pretty reports.

Simplifying code with functions

```
assert( abs( cumulativeNormal(-x) + ...
          cumulativeNormal(x) - 1 ) <0.001;
```

This is hard to understand.

```
function assertApproxEqual( x, y, tolerance )
    assert ( abs(x-y)< tolerance );
end
```

```
assertApproxEqual ( cumulativeNormal(-x), ...
    1- cumulativeNormal(x), 0.001 );
```

Passing functions to functions

In the code below, `f` is a real valued function.

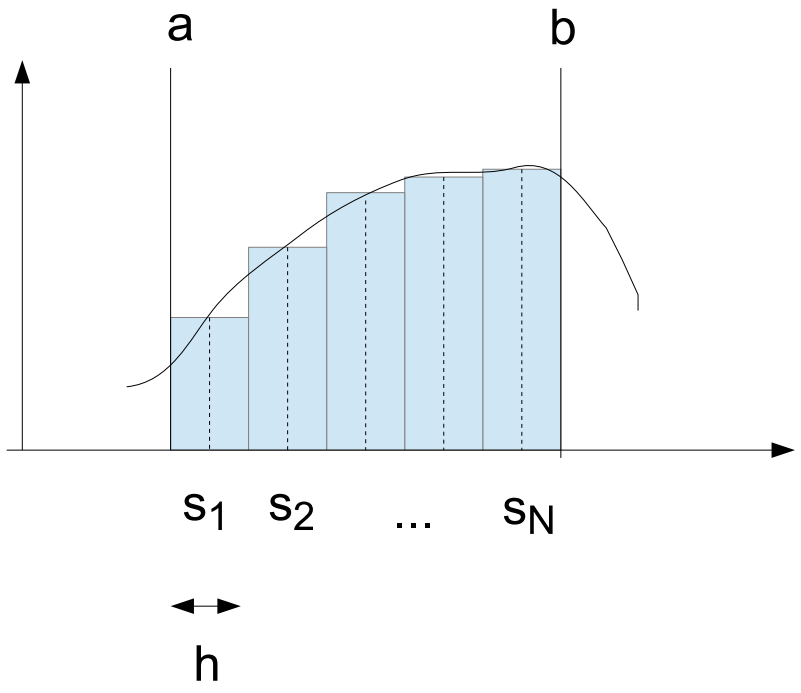
```
function [r]= integrateNumerically(f,a,b,N)
h = (b-a)/N;
s = a + ((1:N) - 0.5) * h;
r = h * sum( f(s) );
end
```

```
integrateNumerically( @sin, 0, 1, 1000 );
```

Graphical interpretation

$$
\begin{aligned}
h &= (b-a)/N \\
s_n &= a + (n - 1/2)h
\end{aligned}
$$

$$
\int_a^b f(s)\,\mathrm{d}s \approx h \sum_1^N f(s_n).
$$



The MATLAB and the maths

```
function [r]= integrateNumerically(f,a,b,N)
h = (b-a)/N;
s = a + ((1:N) - 0.5) * h;
r = h * sum( f(s) );
end
```

Approximate f with N rectangles to compute integral. Define

$$\begin{aligned} h &= (b-a)/N \\ s_n &= a + (n - 1/2)h \end{aligned}$$

then

$$\int_a^b f(s)\,\mathrm{d}s \approx h \sum_1^N f(s_n).$$

Nested functions

Compute

$$\int_0^1 (s^3 + 2s^2)\,\mathrm{d}s$$

```
function result=answerProblem()

function r = integrand( s )
  r = s.^(3) + 2 .* s.^2;
end

NSteps = 1000;
result = integrateNumerically( @integrand, 0, 1, NSteps);

end
```

Nested functions

```
function result=cumulativeNormalVersion2( x )

function r = integrand( s )
  r = s.^(-2) .* exp( -(( x + 1 - 1./s).^2 )/2 );
end

NSteps = 1000;
result = 1/sqrt(2*pi) ...
  * integrateNumerically( @integrand, 0, 1, NSteps);

end
```

You don't have to write a separate file. This function `r` actually depends sneakily upon the variable `x`.

Homework

Write a function `integrateFromMinusInfinity` so that this works:

```
function result=cumulativeNormalVersion3( x )

function r = integrand( s )
  r = exp( -s.^2/2 );
end

NSteps = 1000;
result = 1/sqrt(2*pi) ...
  * integrateFromMinusInfinity( @integrand, x, NSteps);

end
```

What have we done

We started with a very complex function that does too much at once. We divided it into three specialist functions

- `integerateNumerically`. This is the expert in the rectangle rule.
- `integrateFromMinusInfinity`. This is the expert in the substitution needed to change an infinite integratl to a finite one.
- `cumulativeNormal`. This simply states that the cdf is the integral of the pdf.

Testing revisited

How can you test:

- `integrateNumerically`?
- `integrateFromMinusInfinity`?
- `cumulativeNormal`

An example test

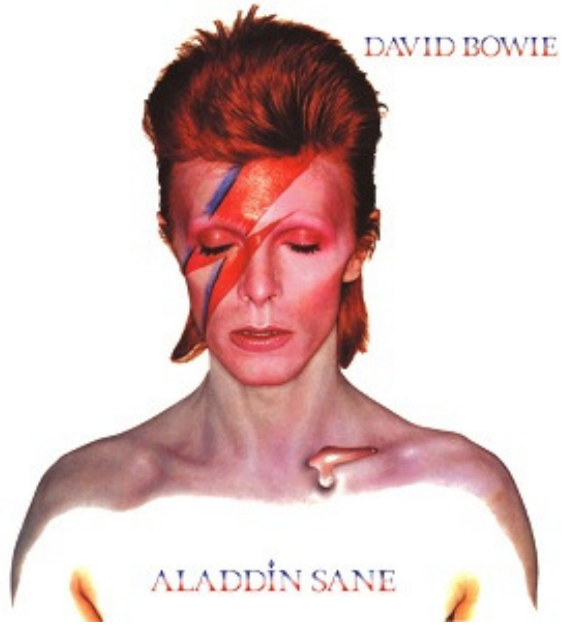
```
function testIntegrateNumerically()

actual = integrateNumerically( @sin, 0, pi, 1000 );
expected = 2.0;
assertApproxEqual( actual, expected, 0.01);

end
```

Function summary

- Divide your code into small, easy to understand functions.
- Write tests for EVERY function.
- Don't throw your tests away. Keep them forever.
- Use `@` for passing functions as arguments.
- Used nested functions for "temporary" functions.

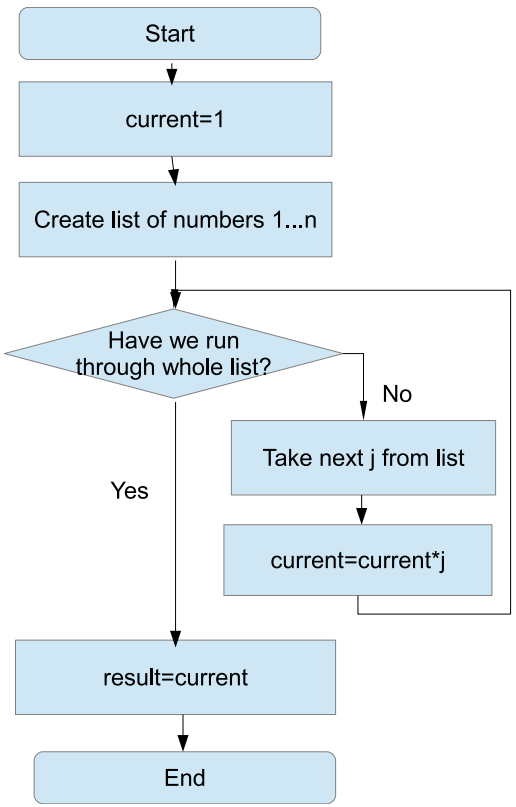
Flow of control

My first computer program




```
function bowieIsCool ()
    for j =1:100
        disp (j);
        disp ('Bowie is cool!');
    end
end
```

A more adult program




```
function result = computeFactorial( n )
    current = 1;
    for j=1:n
        current = current * j;
    end
    result = current;
end
```

With added printout

```
function result = computeFactorial( n )
current = 1;
disp( 'The initial value of current' );
disp( current );
for j=1:n
    disp( 'The value of j');
    disp( j );
    disp( 'The old value of current');
    disp( current );
    current = current * j;
    disp( 'The new value of current');
    disp( current );
end
disp( 'The final value of current');
result = current;
end
```

Counting backwards

```
function launchRocket()
for j=10:-1:1
      disp(j);
end
disp('Blast off');
end
```

Counting backwards again

```
function launchRocket()
for number=10:-1:1
      disp(number);
end
disp('Blast off');
end
```

You can use any variable you like in place of `number`

```
function digitsOfPi()
nums = [ 3 1 4 1];
for j=nums
      disp(j);
end
end
```

You can use any vector in a `for` loop.

If statements

```
function max = maximum( a, b )

if a>b
    disp('a is bigger');
    max = a;
else
    disp('b is bigger');
    max = b;
end

disp('The maximum value is:');
disp( max );

end
```

Testing equality

```
function isValueSeven( value )
if value==7
    disp('The value is seven');
else
    disp('The value is not seven');
end
end
```

Use ~= for not equals.

>=, <=, > and < are all obvious.

Complex tests

|| means OR and && means AND.

```
function isValue3Or7( value )

if value==3 || value==7
    disp('The value is either 3 or 7');
else
    disp('The value is neither 3 nor 7');
end

end
```

Even more complex tests

~ means NOT. This code prints out test passed if

- *a* and *b* are both positive
- or *b* is not equal to 7

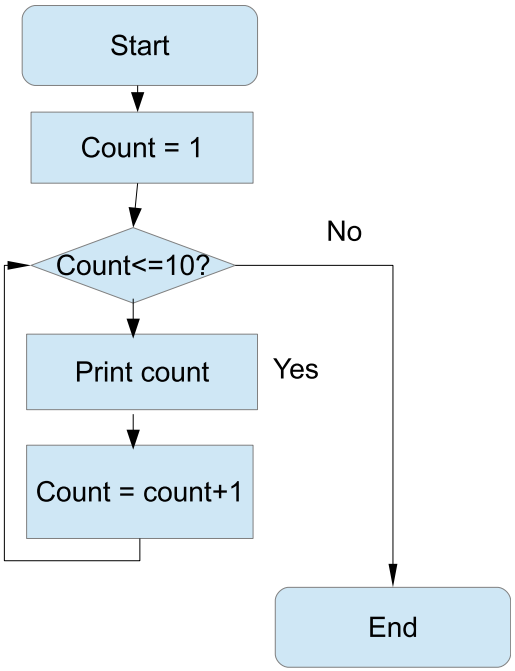
```
function complexTest( a, b )
if (a>0 && b>0) || ~(b==7)
    disp('Test passed');
else
    disp('Test failed');
end
end
```

Note the brackets!

The full syntax for if

```
if test1
    statements
elseif test2
    statements
elseif test3
    statements
elseif test4
    ...
else
    statements
end
```

Counting with a while loop




```
function countUsingWhile()
count = 1;
while count<=10
    disp(count);
    count = count + 1;
end

end
```

In this case, the `for` loop was easier.

While loops

```
while conditionIsTrue
    statements
end
```

A function to find the next prime

```
function prime2=nextPrime(prime1)
current = prime1+1;
while ~isprime(current)
    current = current+1;
end
prime2 = current;
end
```

Notice that this time, the `while` loop does something a `for` loop can't achieve.

Accessing individual cells of a matrix

Reading:

```
A = [ 1 2 3 4; 5 6 7 8];
A(2,3)
```

Writing:

```
A = [ 1 2 3 4; 5 6 7 8];
A(2,3) = -7;
```

Matrices grow as needed

```
A = [ 1 2 3 4; 5 6 7 8];
A(3,6) = -7;
```

changes `A` to be the matrix:

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 0 & 0 \\ 5 & 6 & 7 & 8 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -7 \end{pmatrix}$$

Accessing submatrices

$$A = \begin{pmatrix} 1 & 2 & 3 & 4 & 0 & 0 \\ 5 & 6 & 7 & 8 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -7 \end{pmatrix}$$

- `A(1:3,4)` The fourth column
- `A(2,1:6)` The second row
- `A(1:2,1:2)` The 2×2 submatrix in the top left
- `A(1:end,4)` The fourth column
- `A(2,1:end)` The second row

Using a single index

$$A = \begin{pmatrix} 1 & 6 & 11 \\ 2 & 7 & 12 \\ 3 & 8 & 13 \\ 4 & 9 & 14 \\ 5 & 10 & 15 \end{pmatrix}$$

Then $A(i) = i$.

Putting it all together

Example

Without using the `sum` function, write a function called `mySum` which takes a vector `x` as parameter and adds up all the cells of `x`.

We use a variable `total` to store our working and iterate through all the elements of `x` using a `for` loop increasing `total` as we go:

```
function [total] = mySum( x )
total = 0;
for j=1:length(x)
    total = total + x(j);
end
end
```

Primes

Example

Write a function `primesUpTo` that takes a parameter `n` and returns a vector containing all the prime numbers less than `n`.

```
function primes = primesUpTo( n )
counter = 1;
for j=2:n
    if (isprime(j))
        primes( counter ) = j;
        counter = counter + 1;
    end
end
end
```

Initializing primes

How to stop MATLAB complaining:

```
function primes = primesUpTo( n )
primes = zeros( 1, n );
counter = 1;
for j=2:n
    if (isprime(j))
        primes( counter ) = j;
        counter = counter + 1;
    end
end
primes = primes(1, 1:(counter-1));
end
```

Exercises

★ Write a function `myProd` to compute the product of all the elements in a vector.

★ Write a function to find the maximum value in a vector. You are not allowed to use the MATLAB `max`, `min` or `sort` functions!

Logical values

A Boolean value is either `true` or `false`

- `true` is printed as `1`
- `false` is printed as `0`

You can do arithmetic with Booleans. `(1==2)*7 + (2*2==4)*8`

Positive interpreted as true

```
if 3
    disp('3 is non-zero');
end
```

Comparison operators on matrices

You can use >=, == etc. with matrices

```
v = [-3 -2 -1 0 1 2 3];
isPositive = v > 0;
disp( isPositive );
```

`isPositive` is a matrix of booleans.

```
[0 0 0 0 1 1 1]
```

Matrix programming

- `for` loops are slow in MATLAB
- ”Matrix programming” = getting rid of unecessary loops. Also known as "vectorization".
- There are tricks to matrix programming code. Experienced programmers in other languages will need to learn them too.

Matrix programming example

```
v = [-3 -2 -1 0 1 2 3];
isPositive = v > 0;
disp( isPositive );
```

```
v = [-3 -2 -1 0 1 2 3];
isPositive = zeros( 1, length(v));
for j=1:length(v)
    isPositive(j) = v(j) > 0;
end
disp( isPositive );
```

Matrix programming Tip 1

Use MATLAB's built in functions if possible

- Use `sum` rather than write a `for` loop to sum variables
- Use `normcdf` rather than our homegrown `cumulativeNormal`

Matrix programming Tip 2

Make your functions work with vectors. So replace

```
function interest = computeInterest( P, r, t )
interest = P * (exp( r * t )-1);
end
```

With

```
function interest = computeInterest( P, r, t )
interest = P .* (exp( r .* t )-1);
end
```

Matrix programming Tip 3

How do you change a loop containing an if statement to a matrix calculation?

```
function netProfit = ...
          computeNetProfit( earnings, costs, tax )
grossProfit = earnings-costs;
taxPayable = zeros(length( grossProfit), 1 );
for j=1:length(grossProfit)
    if (grossProfit(j)>0)
        taxPayable(j) = tax * grossProfit;
    end
end
netProfit = grossProfit - taxPayable;
end
```

Matrix Programming Tip 3 (cont)

Like this:

```
          computeNetProfit( earnings, costs, tax )
grossProfit = earnings-costs;
isTaxPayable = grossProfit>0;
taxPayable = isTaxPayable .* grossProfit .* tax;
netProfit = grossProfit - taxPayable;
end
```

Remember that `isTaxPayable` takes the values 1 or 0.

```
if test1
    v = value1;
elseif test2
    v = value2;
else
    v = value3;
end
```

Can be replaced with

```
v = test1 * value1 ...
    + (~test1)*( test2*value2 + (~test2)*value3 );
```

Summary

- We have seen how to use `for` loops
- We have seen how to use `if` statements
- We have seen how to use `while` statements
- We have seen how you can replace `if` statements using arithmetic on boolean values.
- Matrix programming is an optimization technique. It is more important that your code works than that it is fast. But if its taking an hour to run...

Homework

Worksheet 2.
If you are completely new to programming this week's homework is very important.