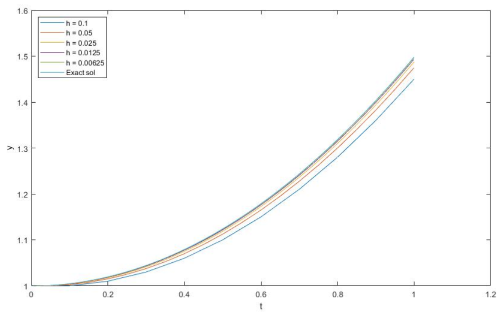
COSC2500 Assignment 4

Ryan White
s4499039

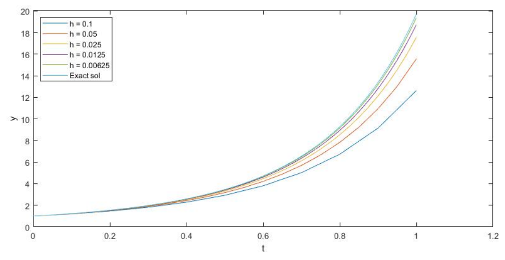
12th of October 2020

R4.1

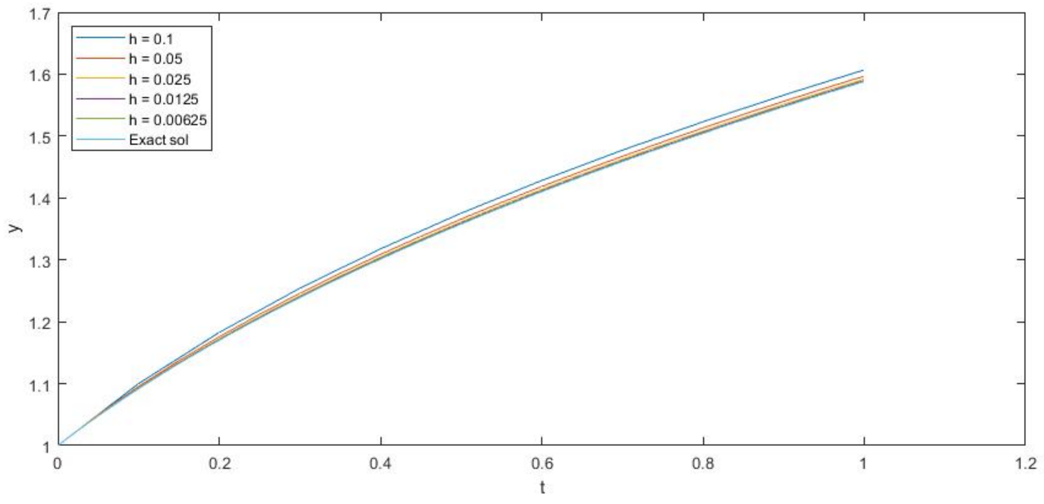
The comparison of Euler's Method for different step sizes against the exact solution of various IVPs is shown in Figure 1.



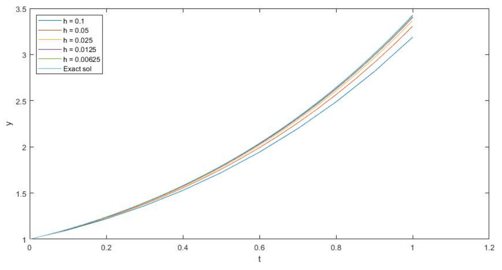
(a) Plotted Solutions for $y' = t$ with varying step sizes



(b) Plotted Solutions for $y' = 2(t+1)y$ with varying step sizes



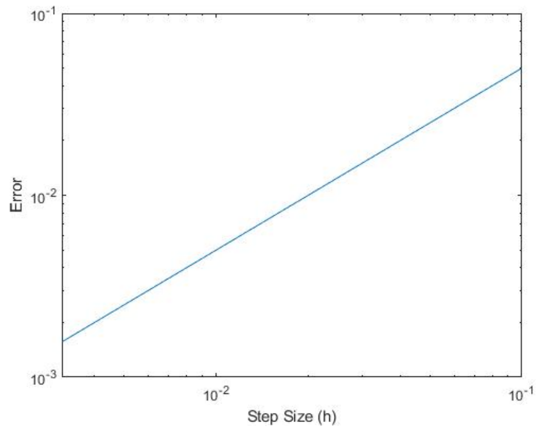
(c) Plotted Solutions for $y' = 1/y^2$ with varying step sizes



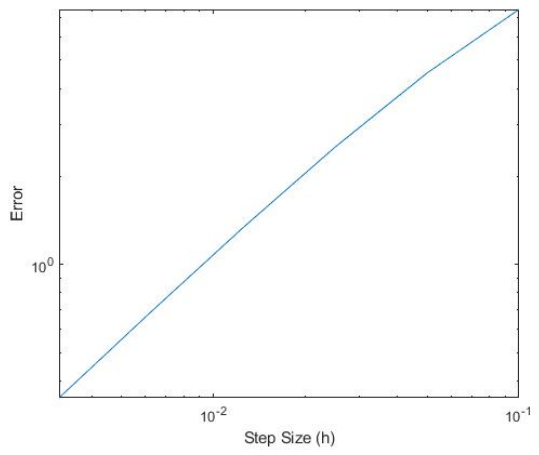
(d) Plotted Solutions for $y' = t + y$ with varying step sizes

Figure 1: Comparisons of Euler Method vs Exact Solution for Varying IVPs and Step Sizes

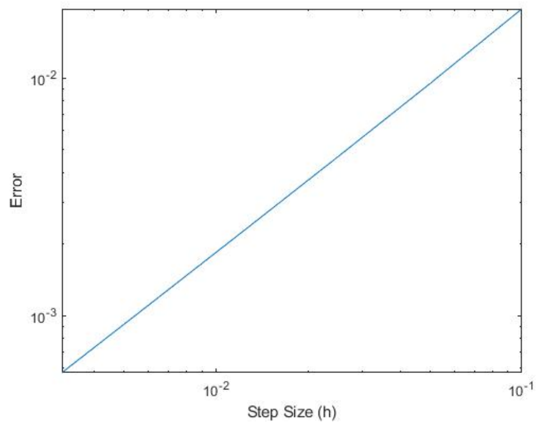
The evident errors between the approximate solutions and the exact solution is shown in Figure 2.



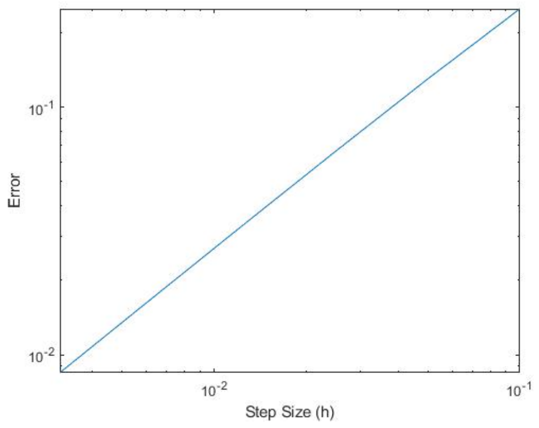
(a) Error of Euler's Method for $y' = t$ with varying step sizes



(b) Error of Euler's Method for $y' = 2(t + 1)y$ with varying step sizes



(c) Error of Euler's Method for $y' = 1/y^2$ with varying step sizes



(d) Error of Euler's Method for $y' = t + y$ with varying step sizes

Figure 2: Comparisons of Error in Euler Method for Varying IVPs and Step Sizes

As can be seen, error is consistently minimised as step size gets smaller.

R4.2

The decision-tree for choosing an appropriate MATLAB ODE solver is shown in Figure 3.

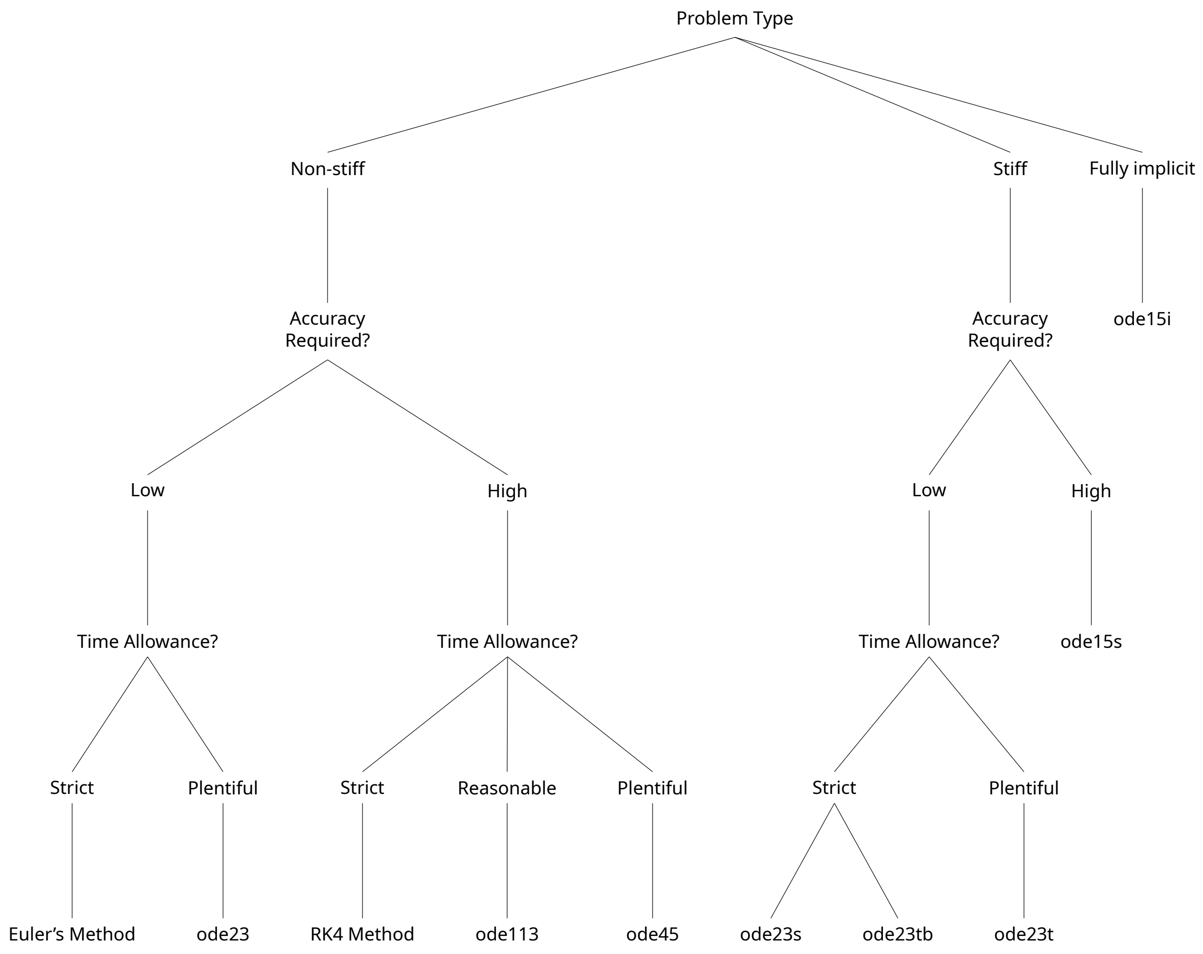
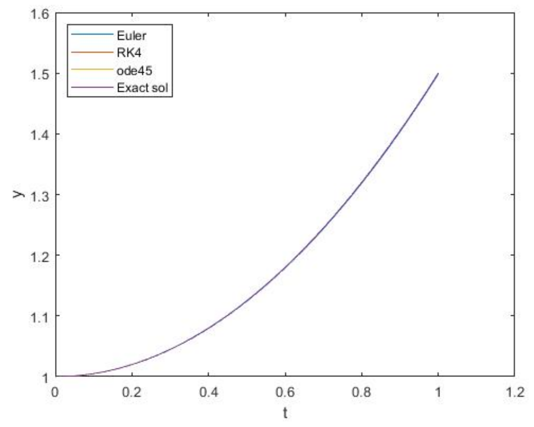


Figure 3: Decision-Tree for Appropriate ODE Solution Method

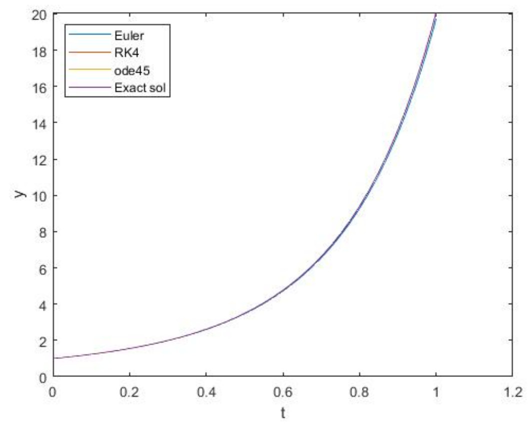
The Euler and RK4 (4th order Runge-Kutta) methods aren't built-in to MATLAB by default, and were provided by Sauer (3rd Edition, 2018), and the course notes respectively.

R4.3

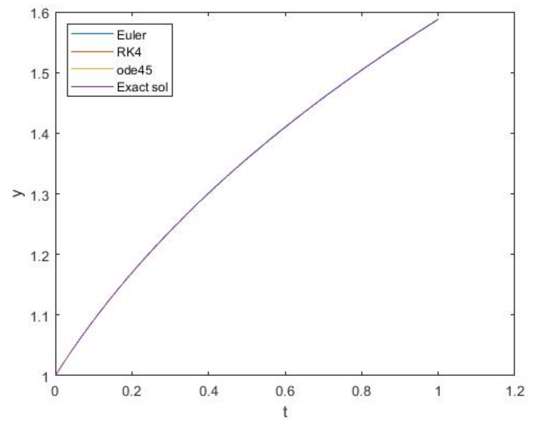
The plots of the different IVPs in terms of their respective function is shown in Figure 4.



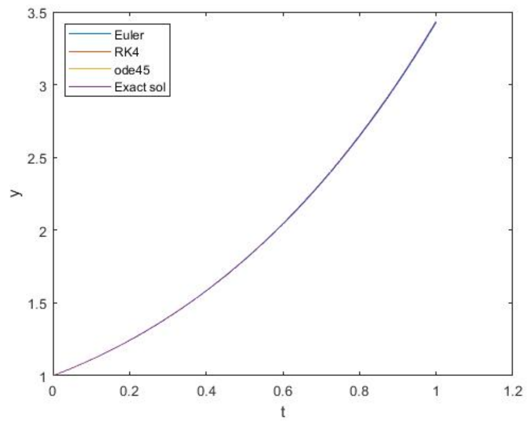
(a) Plotted Solutions for $y' = t$ with fixed step size $h = 0.1 \times 2^{-5}$



(b) Plotted Solutions for $y' = 2(t+1)y$ with fixed step size $h = 0.1 \times 2^{-5}$



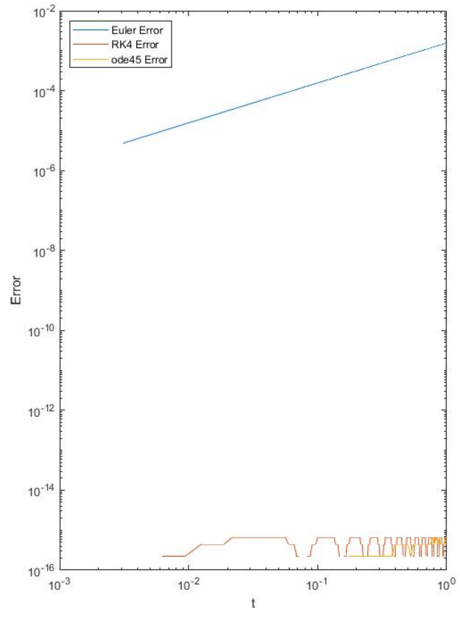
(c) Plotted Solutions for $y' = 1/y^2$ with fixed step size $h = 0.1 \times 2^{-5}$



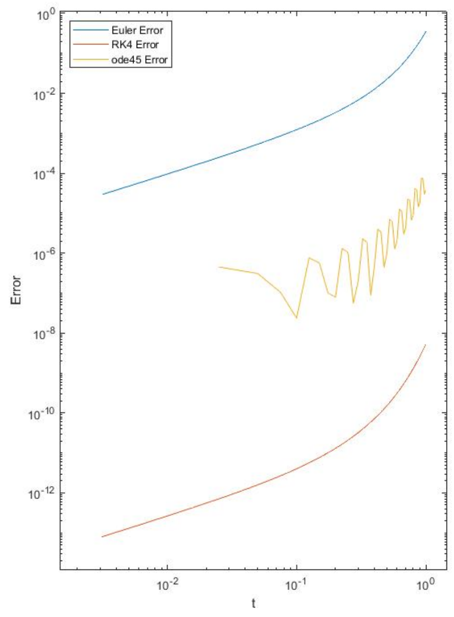
(d) Plotted Solutions for $y' = t + y$ with with fixed step size $h = 0.1 \times 2^{-5}$

Figure 4: Comparisons Euler Method, Runge-Kutta, ode45 and Exact Solutions for Varying IVPs

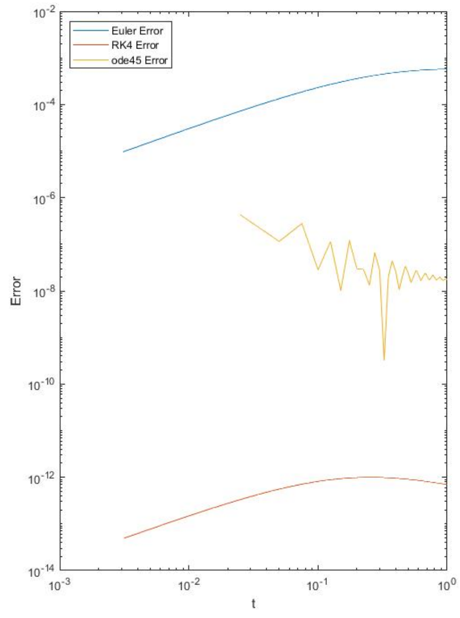
Clearly, relative error between the methods is difficult to discern at this step size. However, difference in accuracy is more easily seen in Figure 5, where the error for each method is compared vs value of t.



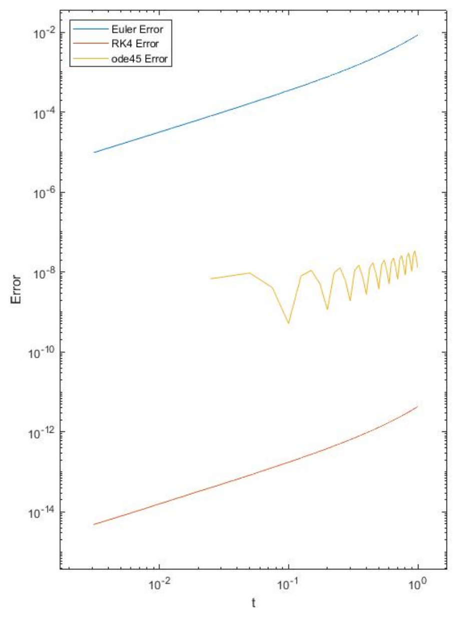
(a) Plotted Solutions for $y' = t$ with fixed step size $h = 0.1 \times 2^{-5}$



(b) Plotted Solutions for $y' = 2(t+1)y$ with fixed step size $h = 0.1 \times 2^{-5}$



(c) Plotted Solutions for $y' = 1/y^2$ with fixed step size $h = 0.1 \times 2^{-5}$



(d) Plotted Solutions for $y' = t + y$ with with fixed step size $h = 0.1 \times 2^{-5}$

Figure 5: Comparisons Euler Method, Runge-Kutta, ode45 and Exact Solutions for Varying IVPs

It is clear that Euler's Method is consistently the least accurate of the 3, with the Runge-Kutta usually the most accurate, bar the solution for $y' = t$ where it is similar in error to the `ode45` method.

R4.4

As per the function description, `ode45` is an implementation of the explicit Runge-Kutta (RK5) method. In the main loop, the function begins by choosing the appropriate step size, `h`. Once the program is underway, if the current step is within 10% of the final t-value, an `if` statement increases the size of `h`. The program then initiates a `while` loop to advance the solution one step while the program has not failed. In the loop, the error is calculated, and the step is only classified as a failure if the calculated error exceeds the tolerance error, `rtol`. An `if` statement then proceeds to lower the step size until the calculated error is within bounds, and the `while` loop continues on. Once the step is successful, the loop is broken, and the total number of steps is increased by one. This repeats until all steps up to the interval end have been computed.

The `ode45` code is fairly advanced, albeit time inefficient. Since it features adaptive step-size, the function is unable to completely pre-allocate array sizes and in doing so requires a lot of time and power to update the array dimensions on each step. Despite this, it seems very efficient in terms of consistently producing solutions within some predefined error tolerance. This behaviour is clearly seen in Figure 5, where the `ode45` error jumps up and down as it adjusts step-size to remain within tolerated error.

1 Appendices

1.1 R4.1 Matlab Code

```
ODEs = {@(t,y) t, @(t,y) 2*(t+1)*y, @(t,y) 1 / y^2, @(t,y) t + y};
sols = {@(t) 1./2 .* (t.^2 + 2), @(t) exp(t .* (t + 2)), @(t) (3.*t+1).^(1./3), @(t) -t + 2.*
    exp(t) - 1};
diff = zeros(5, 1);
for k = 0:5
   steps(k+1) = 0.1 * 2^(-k);
end
figure
for k = 0:5
   h = steps(k+1);
   [t, y] = euler1([0, 1], 1, (1 / h));
   plot(t, y);
   hold on
   diff(k + 1) = abs(sols{4}(1) - y(end));
end

plot([0:1:0.1 * 2^(-5)], sols{4}(t));
legend('h = 0.1', 'h = 0.05', 'h = 0.025', 'h = 0.0125', 'h = 0.00625', 'Exact sol', 'Location',
    'northwest');
xlabel('t');
ylabel('y');

figure
loglog(steps, diff);
xlabel('Step Size (h)');
ylabel('Error');
```

```
function [t, y] = euler1(inter, y0, n)
    t(1) = inter(1); y(1) = y0;
    h = (inter(2) - inter(1)) / n;
    for i = 1:n
        t(i + 1) = t(i) + h;
        y(i + 1) = eulerstep(t(i), y(i), h);
    end
end

function y = eulerstep(t, y, h)
    y = y + h * ydot(t, y);
end

function z = ydot(t, y)
    z = t + y;
end
```

1.2 R4.3 Matlab Code

```
ODEs = {@(t,y) t, @(t,y) 2*(t+1)*y, @(t,y) 1 / y^2, @(t,y) t + y};
sols = {@(t) 1./2 .* (t.^2 + 2), @(t) exp(t .* (t + 2)), @(t) (3.*t+1).^(1./3), @(t) -t + 2.*
     exp(t) - 1};
diff = zeros(5, 1);
for k = 0:5
    steps(k+1) = 0.1 * 2^(-k);
end
figure
k = 5;
h = steps(k+1);
[te, ye] = euler1([0, 1], 1, (1 / h));
[trk, yrk, lastrk] = RK4(ODEs{4}, [0, 1], 1, h);
plot(te, ye);
hold on
plot(trk, yrk);

[to, yo] = ode45(ODEs{4}, [0, 1], 1);
plot(to, yo);
for k = 1:length(te)
    rk4diff(k) = abs(sols{4}(trk(k)) - yrk(k));
    diff(k) = abs(sols{4}(te(k)) - ye(k));
end
for k = 1:length(to)
    odeerr(k) = abs(sols{4}(to(k)) - yo(k));
end

plot(te, sols{4}(te));
legend('Euler', 'RK4', 'ode45', 'Exact sol', 'Location', 'northwest');
xlabel('t');
ylabel('y');

figure
loglog(te, diff);
hold on
loglog(trk, rk4diff);
loglog(to, odeerr);
xlabel('t');
ylabel('Error');
legend('Euler Error', 'RK4 Error', 'ode45 Error', 'Location', 'northwest');


function [t, y] = euler1(inter, y0, n)
    t(1) = inter(1); y(1) = y0;
    h = (inter(2) - inter(1)) / n;
    for i = 1:n
        t(i + 1) = t(i) + h;
        y(i + 1) = eulerstep(t(i), y(i), h);
    end
end

function y = eulerstep(t, y, h)
    y = y + h * ydot(t, y);
end

function z = ydot(t, y)
    z = t + y;
end

function [t,y,last] = RK4(f,tspan,yi,dt)
%Standard RK4 algorithm.
%This code is a standard RK4 algorithm with a fixed step-size.
%It can be used to solve explicit first-order ODEs. The local truncation error is O(dt^5).
%"Time" is the name for the x-axis variable.
%
%Input:
%  f is a function handle for the first-order ODE - dy/dt = f(t,y)
%    - f is assumed to be a row vector.
%  tspan is a vector that contains ti and tf, e.g. tspan = [ti,tf]
%  ti is the initial time
%  tf is the final time
%  yi is the initial values for the ODE. yi is assumed to be a vector.
%  dt is the step size
%
%Output:
%  t is a vector of time values.
%  y is the approximate solution curve for the initial value problem of the ODE.
%  last is the last y values - useful if you are using the last values for
%   something
%
%
%Hint - call the function like this:
%   [T,Y] = RK4(f,tspan,yi,dt)
%
%Code written for COSC2500.
%If you use this code, please just reference it as MOD4 RK4 code.
%There is no need to include it in the write-up - it's on blackboard which tutors can read...
%
%

    ti = tspan(1); tf = tspan(2);
    num_steps = ceil((tf-ti)/dt);  %ceil forces it to be an integer
   t = linspace(ti,tf,num_steps+1).'; %creates time vector, then transposes

   if size(yi,2) > 1
       yi = yi.';
   end
   %Initialising y
   y = [yi,zeros(size(yi,1),num_steps)];

   %Application of the RK4 algorithm.
   for n = 1:num_steps
       k1 = dt*f(t(n),y(:,n));
       k2 = dt*f(t(n) + 0.5*dt, y(:,n) + 0.5*k1);
       k3 = dt*f(t(n) + 0.5*dt, y(:,n) + 0.5*k2);
       k4 = dt*f(t(n) + dt, y(:,n) + k3);
       y(:,n+1) = y(:,n) + (1/6)*(k1+ 2*k2 + 2*k3 + k4);
   end
   y = y.';
   last = y(end,:);
end
```