

EMAT20920: Numerical Methods in MATLAB

COURSEWORK ASSESSMENT

Jake Bowhay (UP19056)

All figures in this report have been saved using `saveFigPDF` function as it automatically resizes the paper to the correct size.

Listing 1: `../src/saveFigPDF.m`

```
function saveFigPDF(fileName)
    %saveFigPDF saves open figure as a PDF file
    %
    %Inputs:
    %  fileName = File name to save figure as
    %Usage:
    %  saveFigPDF("polynomial") -> Saves current figure as polynomial.pdf

    % Get current figure handle
    figureHandle = gcf;
    % Resize paper
    set(figureHandle, 'PaperPosition', 3*[0 0 6 4]);
    set(figureHandle, 'PaperSize', 3*[6 4]);
    set(figureHandle, 'PaperUnits', 'centimeters');

    print(fileName, '-dpdf');
end
```

Question 1: Root-finding

- (a) To find how many solutions each equation has in the given domain I will rearrange all the equations to be equal to zero and then look for the zeros of the rearranged equations. As a corollary to the intermediate value theorem, if a function is continuous and changes sign in a bracket then that bracket must contain a zero. So I will plot each of the rearranged equation and I look for appropriate brackets. I will use the `pltFunc` function to plot the functions as it removes values outside a defined limit which prevents MATLAB plotting discontinuous functions as continuous. The limits can then be changed using the property explorer to give a more useful plot.

Listing 2: `../src/q1/pltFunc.m`

```
function pltFunc(f, domain, discountLim)
    %pltFunc plots function f between values of xLim removing any values
    % that are greater than discountLim to prevent MATLAB plotting
    % discontinuous functions as continuous and plots a line of x = 0 to
    % help make any zeros clear
    %
    % Input:
    %  f = function handle to plot
    %  domain = 1x2 vector containing the lower and upper bound of the
    %  domain of f
    %  discountLim = absolute values of the function greater than this are
    %  changed to NaN. Setting to inf will plot all values of the function
    %
```

```

% Usage:
% pltFunc(@(x) 1./x, [-10 10], 5) -> Plots 1/x between -10 and 10
% changing the values where |1/x|>5 to NaN

% Check xlim is the correct dimensions
assert(isequal(size(domain), [1 2]), "domain must be a 1x2 vector")

%% Generate values to plot
x = linspace(domain(1), domain(2));
y = f(x);

% Remove large values of y to prevent MATLAB plotting discontinuous
% functions as continuous
y(abs(y)>discontLim) = NaN;

%% Plot function and line x = 0
plot(x, y, [min(x) max(x)], [0 0], "g-", "LineWidth", 2);
xlabel("x");
ylabel("f(x)");
xlim(domain);
title("Plot of f(x)");
grid on;
end

```

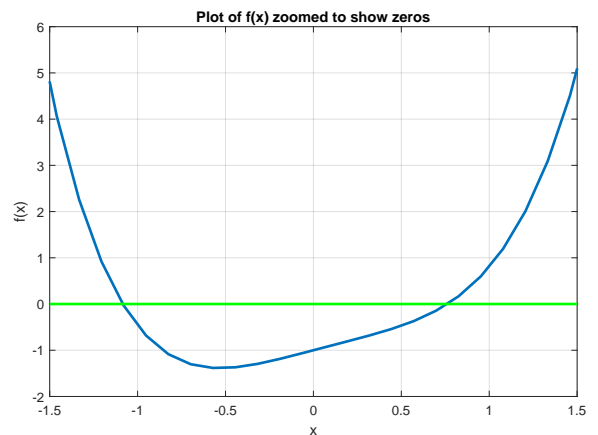
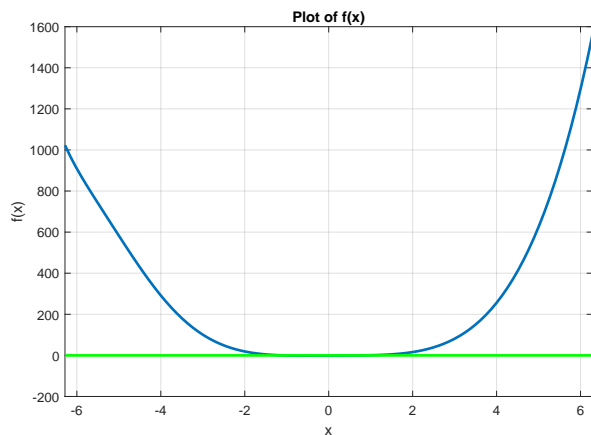
- (i) Rearranging $x^4 = e^{-x} \cos(x)$ gives $f(x) = x^4 - e^{-x} \cos(x)$.

Listing 3: ../src/q1/Q1a.i.m

```

f = @(x) x.^4 - exp(-x).*cos(x);
pltFunc(f, [-2*pi 2*pi], inf);

```



The second zoomed in plot shows there are two zeros in the given domain. The first zero can be bracketed by the interval $[-1.5, -1]$ as $f(-1.5) = 4.7455$ and $f(-1) = -0.4687$ so since the function is continuous and there is a change of sign this bracket must contain a zero. Like wise the second root can be bracketed by the interval $[0.5, 1]$ as $f(0.5) = -0.4698$ and $f(1) = 0.8012$.

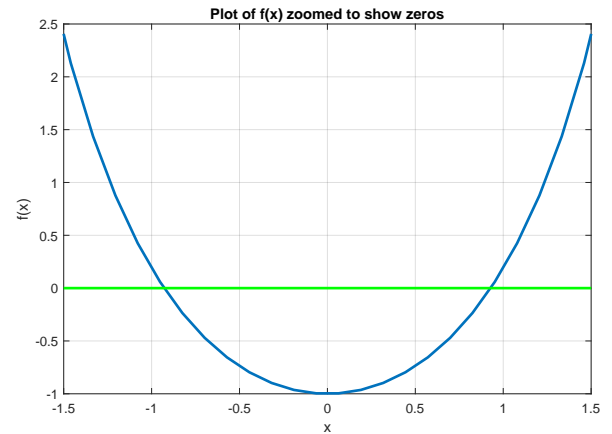
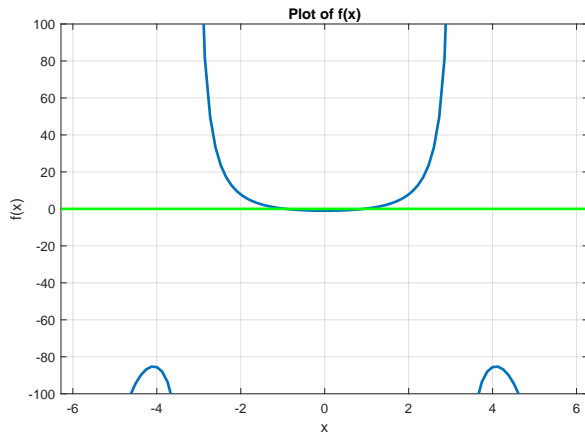
- (ii) Setting $f(x) = \frac{x^3}{\sin(x)} - 1$.

Listing 4: ../src/q1/Q1a.ii.m

```

f = @(x) (x.^3)./sin(x) - 1;
pltFunc(f, [-2*pi 2*pi], 500);

```

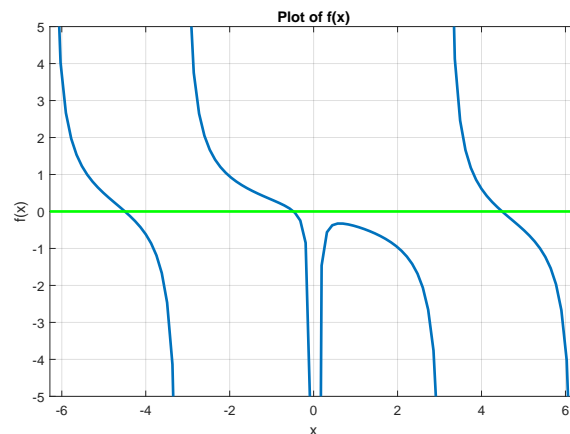


The second plot show there are two roots. The first root can be bracketed by the interval $[-1, -0.5]$ as $f(-1) = 0.1884$ and $f(-0.5) = -0.7393$ and $f(x)$ is continuous in this bracket. Likewise, the second root can be bracketed by the interval $[0.5, 1]$ as $f(0.5) = -0.7393$ and $f(1) = 0.1884$.

- (iii) Rearranging $\cot(x) = \frac{25}{25x-1}$ gives $f(x) = \cot(x) - \frac{25}{25x-1}$.

Listing 5: `../src/q1/Q1a_iii.m`

```
f = @(x) cot(x) - 25./(25*x - 1);
pltFunc(f, [-2*pi 2*pi], 30);
```

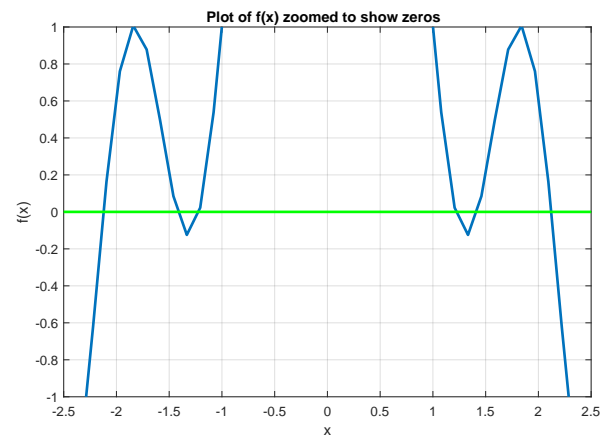
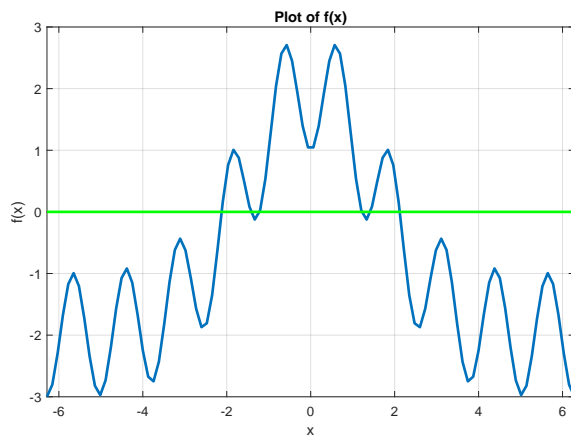


The plot shows that the equation has three solutions. The first can be bracketed by the interval $[-5, -4]$ as $f(-5) = 0.4942$ and $f(-4) = -0.6162$. The second solution can be bracketed by the interval $[-1, -0.1]$ as $f(-1) = 0.3194$ and $f(-0.1) = -2.8238$. The third solution can be bracketed by the interval $[4, 5]$ as $f(4) = 0.6112$ and $f(5) = -0.4974$. $f(x)$ is continuous in each of the bracketing intervals.

- (iv) Rearranging $4e^{-x^2/5} = \cos(5x) + 2$ gives $f(x) = 4e^{-x^2/5} - \cos(5x) - 2$.

Listing 6: `../src/q1/Q1a_iv.m`

```
f = @(x) 4*exp(-x.^2/5) - cos(5*x) - 2;
pltFunc(f, [-2*pi 2*pi], inf);
```



The second plot shows that the equation has 6 solutions. The bracketing intervals are shown in the table below.

$[a, b]$	$f(a)$	$f(b)$
$[-2.5, -2]$	-1.8518	0.6364
$[-1.5, -1.25]$	0.2039	-0.0730
$[-1.25, -1]$	-0.0730	0.9913
$[1, 1.25]$	0.9913	-0.0730
$[1.25, 1.5]$	-0.0730	0.2039
$[2, 2.5]$	0.6364	-1.8518

- (b) The bisection method is used by calling the the `bisectRoot` function.

Listing 7: `../src/q1/bisectRoot.m`

```
function [sol, i, err] = bisectRoot(f, a, b, tol)
    %bisectRoot Use the bisection method to find roots of the function f
    % bracketed within the intervals [a, b].
    %
    %Inputs:
    % f = function handle to function whose root is to be found
    % a = 1*n array containing all the lower ends of the brackets
    % where n is the number of roots
    % b = 1*n array containing all the lower ends of the brackets
    % where n is the number of roots
    % tol = absolute error tolerance with which to find the root;
    % Iteration terminates when the root is known to within +/- tol
    %
    %Outputs:
    % sol = 1*n array of of roots
    % i = 1*n array of the number of iterations required to find the nth
    % root
    % err =
    %
    %Usage:
    % [r, i, err] = bisect(@(x) x.^2 - 4, 1, 3, 5e-9) -> returns the
    % approximation to root of  $x^2 - 4 = 0$  within  $[1, 3]$ , the number of
    % iterations required to find the root and the final absolute error
    %
    % check if all intervals are correctly defined
```

```

assert(isequal(size(a), size(b)),...
    "Must be an equal number of upper and lower bounds");

% check whether f changes sign
assert(all(sign(f(a)) ~= sign(f(b))),...
    'f(a) and f(b) should have opposite sign');

% initialise variables
% iteration counter
i = zeros(size(a));
% current solution estimate
sol = (a + b)/2;
% previous solution estimate
sol_old = Inf;
% absolute error
err = Inf;
withinTol = zeros(size(a));

% bisection algorithm:
% at each iteration, find the half-interval that contains a sign change
% and relabel the endpoints appropriately
while any(~withinTol)
    i(~withinTol) = i(~withinTol) + 1;
    sol_old = sol;
    mid = (a + b)/2;

    % mid point is a root
    exactRoot = f(mid) == 0;
    sol(exactRoot) = mid(exactRoot);
    err(exactRoot) = 0;
    withinTol(exactRoot) = true;

    % solution is in first half of interval and mid point not a root
    firstHalf = (sign(f(a)) ~= sign(f(mid))) & ~exactRoot;
    b(firstHalf) = mid(firstHalf);

    % solution is in second half of interval and mid point not a root
    secondHalf = (sign(f(a)) == sign(f(mid))) & ~exactRoot;
    a(secondHalf) = mid(secondHalf);

    % update solutions and errors values that aren't within tolerance
    sol(~withinTol) = (a(~withinTol) + b(~withinTol))/2;
    err(~withinTol) = abs(sol(~withinTol) - sol_old(~withinTol));
    withinTol(err < tol) = true;
end
end

```

- (i) Solutions to $f(x) = x^4 - e^{-x} \cos(x) = 0$ $x \in [-2\pi, 2\pi]$.

Listing 8: `../src/q1/Q1b.i.m`

```

f = @(x) x.^4 - exp(-x).*cos(x);
a = [-1.5 0.5];
b = [-1 1];
[r, i, err] = bisectRoot(f, a, b, [5e-8 5e-9])

```

Note the two different tolerances since one root is an order of magnitude larger so requires one less decimal place of accuracy to be accurate to 8 significant figures.

- (ii) Solutions to $f(x) = \frac{x^3}{\sin(x)} - 1 = 0$ $x \in [-2\pi, 2\pi]$.

$[a, b]$	Root	# Iterations
$[-1.5, -1]$	-1.0843597	23
$[0.5, 1]$	0.76221107	26

Listing 9: ../src/q1/Q1b.ii.m

```
f = @(x) (x.^3)./sin(x) - 1;
a = [-1 0.5];
b = [-0.5 1];
[r, i, err] = bisectRoot(f, a, b, 5e-9)
```

$[a, b]$	Root	# Iterations
$[-1, -0.5]$	-0.92862631	26
$[0.5, 1]$	0.92862631	26

(iii) Solutions to $f(x) = \cot(x) - \frac{25}{25x-1} = 0$ $x \in [-2\pi, 2\pi]$.

Listing 10: ../src/q1/Q1b.iii.m

```
f = @(x) cot(x) - 25./(25*x - 1);
a = [-5 -1 4];
b = [-4 -0.1 5];
[r, i, err] = bisectRoot(f, a, b, [5e-8 5e-9 5e-8])
```

$[a, b]$	Root	# Iterations
$[-5, -4]$	-4.4953722	24
$[-1, -0.1]$	-0.47773376	27
$[4, 5]$	4.4914097	24

(iv) Solutions to $f(x) = 4e^{-x^2/5} - \cos(5x) - 2 = 0$ $x \in [-2\pi, 2\pi]$.

Listing 11: ../src/q1/Q1b.iv.m

```
f = @(x) 4*exp(-x.^2/5) - cos(5*x) - 2;
a = [-2.5 -1.5 -1.25 1 1.25 2];
b = [-2 -1.25 -1 1.25 1.5 2.5];
[r, i, err] = bisectRoot(f, a, b, 5e-8)
```

$[a, b]$	Root	# Iterations
$[-2.5, -2]$	-2.1222382	23
$[-1.5, -1.25]$	-1.4255432	22
$[-1.25, -1]$	-1.2145933	22
$[1, 1.25]$	1.2145933	22
$[1.25, 1.5]$	1.4255432	22
$[2, 2.5]$	2.1222382	23

(c) The iterative scheme we asked to implement is called Steffensen's method. This is implemented in the `steffensenRoot` function.

Listing 12: ../src/q1/steffensenRoot.m

```
function [r, n, err] = steffensenRoot(f, x0, tol, nMax)
```

```

%steffensenRoot uses Steffensen's method to find roots of f(x)
% based on an initial guess x0
%
%Inputs:
% f = function handle to function whose root is to be found
% x0 = initial guess of the root to begin iteration at
% tol = absolute error tolerance with which to find the root
% iteration terminates when the root is known to within +/- tol
% nMax = the maximum number of iteration to quit after. Prevents an
% infinite loop if the iterations do not converge
%
%Outputs:
% r = the approximate root of f(x)=0
% n = the number of iterations
% err = 1*n vector of the absolute error after each iteration
%
%Usage:
% [r, n, err] = steffensenRoot(@(x) exp(-x) -x, 0, 5e-9, 50) ->
% returns the approximate roo of  $x^{-x} - x = 0$  after n iterations and
% err the absolute error after each iteration

% set initial guess as first root
xn = x0;
%iteration counter
n = 0;
% preallocate error array
err = Inf(1, nMax);

while all(err > tol) && n < nMax
    n = n + 1;
    xOld = xn;
    % Calculate f(xn) to avoid repeat computation
    fn = f(xn);
    % Calculate next iteration
    xn = xn - fn*(f(xn + fn)/fn - 1)^-1;
    err(n) = abs(xn - xOld);
end

% remove any unused preallocated element in error array
err(isinf(err)) = [];

% check if solution converged
assert(err(end) < tol, "No convergence")

r = xn;
end

```

The following uses this function to find the root of $e^{-x} - x = 0$ and calculate the convergence.

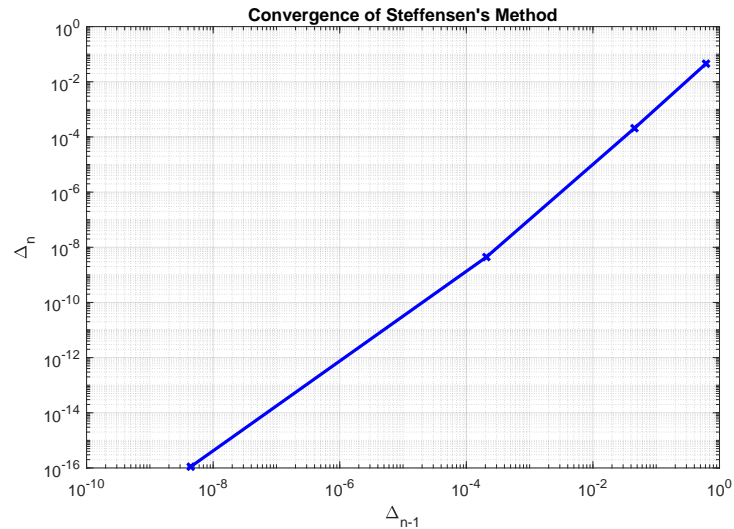
Listing 13: ../src/q1/Q1c.m

```

f = @(x) exp(-x) - x;
[r, n, e] = steffensenRoot(f, 0, 5e-13, 50)
%% Generate plot of convergence
loglog(e(1:end-1),e(2:end), "bx-", "LineWidth", 2);
title("Convergence of Steffensen's Method")
xlabel('\Delta_{n-1}');
ylabel('\Delta_{n}');
grid on;
%% Find order of congerence
polyfit(log(e(1:end-1)), log(e(2:end)), 1)

```

After 5 iterations the root $x = 0.567143290410$ is accurate to 12 decimal places.



The graph shows a straight line which shows $\text{error} \propto \Delta_{n-1}^q$, where q is the gradient of the line. Using the MATLAB function `polyfit` the gradient of the above graph as 1.8 which is close to 2 so Steffensen's method is second order.

- (d) The first step in creating a cobweb plot is to implement a fixed point iteration scheme.

Listing 14: `../src/q1/fixedPointRoot.m`

```
function xn = fixedPointRoot(g, x0, nMax)
    % fixedPointRoot Iteration to find solutions of x = g(x)
    %
    %Inputs:
    %   g = function handle to find the solutions of x = g(x)
    %   x0 = first term of the iteration
    %   nMax = the maximum number of iteration to quit after
    %
    %Output:
    %   xn = the iterative sequence
    %
    %Usage:
    %   xn = fixedPointRoot(@(x) cos(x), 0.75, 100) -> looks for a
    %   root of the equation x - cos(x) = 0, starting with an initial guess
    %   of 0.75.

    % number of iterations
    n = 0;
    % preallocated sequence array and set initial guess as first term
    xn = NaN(1, nMax);
    xn(1) = x0;
    % set initial error
    err = Inf;

    % iterate x -> g(x)
    while n < nMax
        n = n + 1;
        xn(n + 1) = g(xn(n));
        err = abs(xn(n + 1) - xn(n));
    end
```



```

end

% remove any unused elements of the preallocated array
xn(isnan(xn)) = [];

fprintf('\nAfter %d steps root is %-20.14g\n', n, xn(end));
fprintf('Final absolute error is %g\n\n', err);
end

```

Listing 15: ../src/q1/cobwebDiagram.m

```

function cobwebDiagram(g, x0, nMax, a, b)
%cobwebDiagram Creates cobweb diagram for x = g(x) in interval [a,b]
%
%Inputs:
% g = function handle for g(x)
% x0 = initial guess to start iteration
% nMax = number of iterations to complete
% a = lower end of interval [a,b] to plot cobweb diagram over
% b = upper end of interval [a,b] to plot cobweb diagram over
%
%Usage:
% cobwebDiagram(@(x) (x.^5 + 3)/5, 1, 10, 0, 1.5) -> produces a
% cobweb diagram of x = (x^5 + 3)/5 based on an initial guess of 10
% and 10 iterations. This is shown over the interval [0,1.5].

%% get fixed point iteration sequence
xn = fixedPointRoot(g, x0, nMax);

%% generate cobweb diagram

% get values for the line y = x and y = g(x)
x = linspace(a, b);
y = g(x);
y(isinf(y)) = NaN;

% set up figure
hold on;
grid on;
set(gca, "DefaultLineLineWidth", 2);
title("Cobweb plot for fixed point iteration");
xlabel("x");
ylabel("y")
xlim([a b])
ylim([min(x(1), y(1)) max(x(end), y(end))]);

% plot lines y = x and y = g(x)
plot(x, x, "r-", "DisplayName", "y = x");
plot(x, y, "k-", "DisplayName", "y = g(x)");
legend("AutoUpdate", "off");

% plot the steps
plot([xn(1) xn(1)], [0 xn(2)], 'm-');
for i=1:length(xn) - 2
    plot([xn(i) xn(i + 1)], [xn(i + 1) xn(i + 1)], 'm-');
    plot([xn(i + 1) xn(i + 1)], [xn(i + 1) xn(i + 2)], 'm-');
end
end

```

Question 2: Numerical integration and differentiation

- (a) (i) The first expression is Simpson's 3/8 rule and the second is Milne's rule.

Simpson's 3/8 rule can be implemented as follows.

Listing 16: ../src/q2/simpson38.m

```
function simpQuad = simpson38(f, a, b)
    %simpson38 approximates integral of f(x) over interval [a,b] by using
    %Simpson's 3/8 rule
    %
    %Inputs:
    % f = function handle of the integrand f(x)
    % a = lower bound of the interval
    % b = upper bound of the interval
    %
    %Outputs:
    % simpQuad = approximate quadrature
    %
    %Usage:
    % quad = simpson38(@(x) x^2, 0, 0.5) -> returns the approximate
    % intergal of x^2 in the interval [0, 0.5]

    simpQuad = (b - a)/8 .* (f(a) + 3*f((2*a + b)/3) + 3*f((a + 2*b)/3)...
        + f(b));
end
```

And similarly Milne's rule can be implemented.

Listing 17: ../src/q2/milne.m

```
function milneQuad = milne(f, a, b)
    %milne approximates integral of f(x) over interval [a,b] by using
    %Milne's rule
    %
    %Inputs:
    % f = function handle of the integrand f(x)
    % a = lower bound of the interval
    % b = upper bound of the interval
    %
    %Outputs:
    % milneQuad = approximate quadrature
    %
    %Usage:
    % quad = milne(@(x) x^2, 0, 0.5) -> returns the approximate
    % intergal of x^2 in the interval [0, 0.5]

    milneQuad = (b - a)/3 .* (2*f((3*a + b)/4) - f((a + b)/2)...
        + 2*f((a + 3*b)/4));
end
```

However to use the composite version the integral must be broken down into smaller intervals. For example breaking the integral into n intervals gives $\int_a^b f(x)dx = \int_a^{x_1} f(x)dx + \int_{x_1}^{x_2} f(x)dx + \dots + \int_{x_{n-1}}^b f(x)dx$ where $x_i = a + i \cdot \frac{b-a}{n}$. Then each of these smaller integrals can be calculated using either of the methods. The `compositeQuad` function breaks down the integral into smaller intervals before using a Newton-Cotes method of choice to approximate the integral.

Listing 18: ../src/q2/compositeQuad.m

```
function [compQuad, h, err] = compositeQuad(f, i, a, b, tol)
    %compositeQuad amproximate quadrature using a Newton-Cotes method
```

```

%
%Inputs:
% f = function handle of the integrand
% i = function handle of the Newton-Cotes method to use. Must be in
% the format i(f, a, b) where f is the integrand and [a,b] is the
% interval to integrate over
% a = lower bound of the interval
% b = upper bound of the interval
% tol = desired absolute error tolerance
%
%Outputs:
% compQuad = vector of the successive approximates of the
% quadrature where the final entry is the final approximate
% h = vector of the step size used at each approximation
% err = vector of the absolute error at each approximation
%
%Usage:
% compositeQuad(@(x) exp(x), @(f, a, b) (b - a)/2 .* (f(a) + f(b)),...
% 0, 1, 5e-4) -> Estimates the quadrature of e^x in the interval
% [0,1] using the trapezium rule to 3 decimal places

% max number of iterations to prevent infinite loop
nMax = 25;
% iteration counter
n = 1;
% number of subintervals
N = 2;
% preallocate vectors for the error, quadrature and step size
err = inf(1, nMax);
compQuad = NaN(1, nMax);
h = NaN(1, nMax);

% composite algorithm
while all(err > tol) && n < nMax
    % generate step size
    h(n) = (b - a)/N;
    % calculate quadrature using given Newton-cotes method
    compQuad(n) = sum(i(f, a + h(n).*[0:N-1], a + h(n).*[1:N]));
    % calculate absolute error
    try
        err(n) = abs(compQuad(n) - compQuad(n - 1));
    catch
        % prevents error when calculating first error term as no
        % previous approximation to compare against
        err(n) = inf;
    end
    n = n + 1;
    N = N * 2;
end
% removed any used preallocation
err(isinf(err)) = [];
compQuad(isnan(compQuad)) = [];
h(isnan(h)) = [];
end

```

For an example both methods can be used to evaluate $\int_0^5 e^x - x dx$ to 6 decimal places as follows.

Listing 19: ../src/q2/Q2ai_example.m

```
f = @(x) exp(x) - x;
a = 0;
b = 5;
tol = 5e-7;
% using Simpson's 3/8 rule
compositeQuad(f, @simpson38, a, b, tol)
% using Milne's rule
compositeQuad(f, @milne, a, b, tol)
```

Both give the answer to the example as 134.913159.

(ii) test

Question 3: Numerical solution of ODEs

Listing 20: ../src/q3/rhs_projectile.m

```
(a) function dydt = rhs_projectile(t, y, g, mu)
    %rhs_projectile returns column vector of [x,y,u,v]
    dydt = [y(3), y(4), -mu * y(3) * (y(3)^2 + y(4)^2)^0.5, ...
            -g - mu * y(4) * ((y(3)^2 + y(4)^2)^0.5)]';
end
```

(b) (i) Forward Euler

Listing 21: ../src/q3/forwardEulerProjectile.m

```
function [t, y] = forwardEulerProjectile(rhs, tSpan, y0, g, mu, n)
    %forwardEulerProjectile solves the projectiles motion using forward
    %euler
    %
    %Inputs:
    %   rhs = function handle for rhs of ode returning a column vector
    %   [x,y,u,v]
    %   tSpan = vector [a,b] which is the time interval to solve ODE over
    %   y0 = 1*4 vector of initial conditions [x0,y0,u0,v0]
    %   g = value of the acceleration due to gravity
    %   mu = value of the drag parameter
    %   n = the number of steps to split integration over
    %
    %Outputs:
    %   t = column vector of solution times
    %   y = matrix of solutions where each row is the values of each of the
    %   variables at the corresponding value of t in the same fashion as
    %   ode45
    %Usage:
    %   [t,y]=forwardEulerProjectile(@rhs_projectile, [0 5],...
    %   [0 0 31 21]', 9.81, 2.79e-2, 100) -> Solves the projectile ODE from
    %   t=1 to 5 with 100 steps

    t = linspace(tSpan(1), tSpan(end), n + 1);
    % preallocate solution matrix
    y = zeros(numel(t), numel(y0));
    % calculate step size
    h = (tSpan(end) - tSpan(1))/n;

    % parse parameter values to RHS function
    f = @(t,y) rhs(t, y, g, mu);
```

```

% set initial conditions
y(1,:) = y0';

% forward euler method
for i = 1:n
    y(i + 1,:) = y(i,:) + h * f(t(i), y(i,:))';
end

end

```

(ii) 4th-order Runge-Kutta

Listing 22: ../src/q3/rk4Projectile.m

```

function [t, y] = rk4Projectile(rhs, tSpan, y0, g, mu, n)
%rk4Projectile solves the projectile motion using 4th order Runge-Kutta
%
%Inputs:
% rhs = function handle for rhs of ode returning a column vector
% [x,y,u,v]
% tSpan = vector [a,b] which is the time interval to solve ODE over
% y0 = 1*4 vector of initial conditions [x0,y0,u0,v0]
% g = value of the acceleration due to gravity
% mu = value of the drag parameter
% n = the number of steps to split integration over
%
%Outputs:
% t = column vector of solution times
% y = matrix of solutions where each row is the values of each of the
% variables at the corresponding value of t in the same fashion as
% ode45
%Usage:
% [t,y]=rk4Projectile(@rhs_projectile, [0 5],...
% [0 0 31 21]', 9.81, 2.79e-2, 100) -> Solves the projectile ODE from
% t=1 to 5 with 100 steps

t = linspace(tSpan(1), tSpan(end), n + 1);
% preallocate solution matrix
y = zeros(numel(t), numel(y0));
% calculate step size
h = (tSpan(end) - tSpan(1))/n;

% parse parameter values to RHS function
f = @(t,y) rhs(t, y, g, mu);

% set initial conditions
y(1,:) = y0';

for i=1:n
    m1 = f(t(i), y(i,:))';
    m2 = f(t(i) + h/2, y(i,:) + m1 * h/2)';
    m3 = f(t(i) + h/2, y(i,:) + m2 * h/2)';
    m4 = f(t(i) + h, y(i,:) + m3 * h)';
    y(i + 1,:) = y(i,:) + (h/6)*(m1 + 2*m2 + 2*m3 + m4);
end

end

```

(iii) ode45