Department of Engineering Mathematics

EMAT20920: Numerical Methods in MATLAB

COURSEWORK ASSESSMENT

Jake Bowhay (UP19056)

Contents

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
```

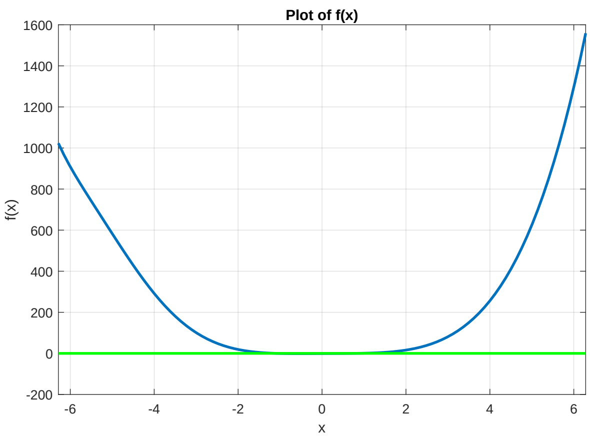
Question 1: Root-finding

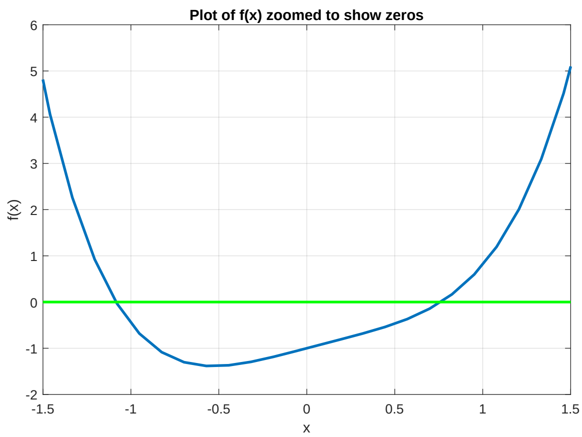
(a) To find how many solutions each equation has in the domain I will rearrange all the equations to be equal to zero and then looks for the roots of the rearranged equations. As a corollary to the intermediate value theorem, if a function is continuous and changes sign in an interval then that bracketing interval must contain a root. So I will plot each of the rearranged equation and look for bracketing intervals that contain a root. I will use the `pltFunc` function (as shown in Listing 30) 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 show more detail if needed.

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

Listing 2: `../src/q1/Q1a_i_funcPlt.m`

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





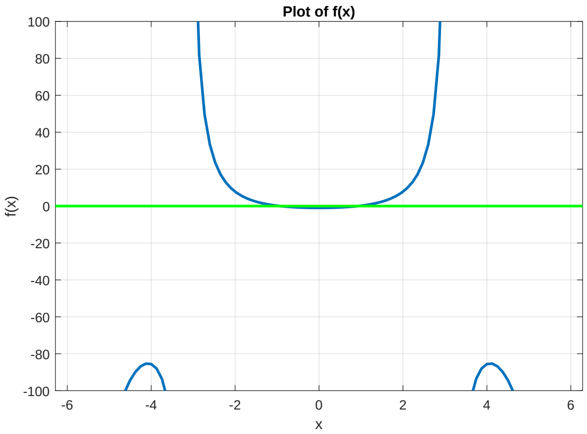
The second zoomed plot shows there are two solutions in the given domain.

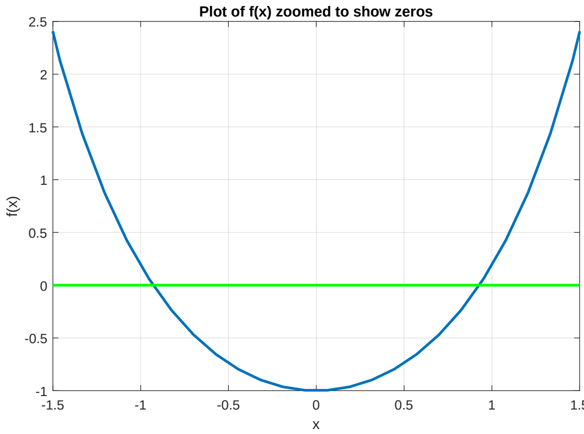
$[a,b]$	f(a)	f(b)	Continuous over $[a,b]$
$[-1.5,-1]$	4.7455	-0.4687	Yes
$[0.5,1]$	-0.4698	0.8012	Yes

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

Listing 3: `../src/q1/Q1a_ii_funcPlt.m`

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





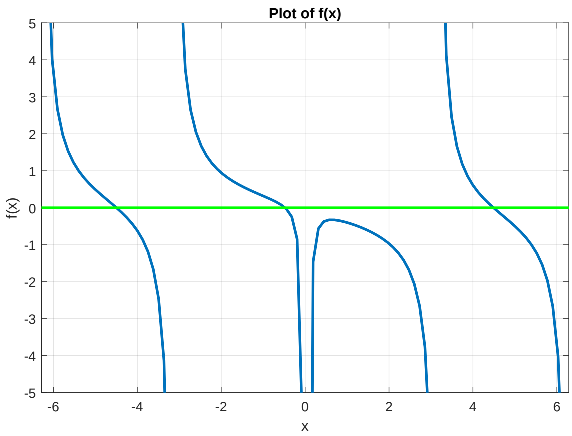
The second plot shows there are two roots.

$[a,b]$	f(a)	f(b)	Continuous over $[a,b]$
$[-1,-0.5]$	0.1884	-0.7393	Yes
$[0.5,1]$	-0.7393	0.1884	Yes

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

Listing 4: `../src/q1/Q1a_iii_funcPlt.m`

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



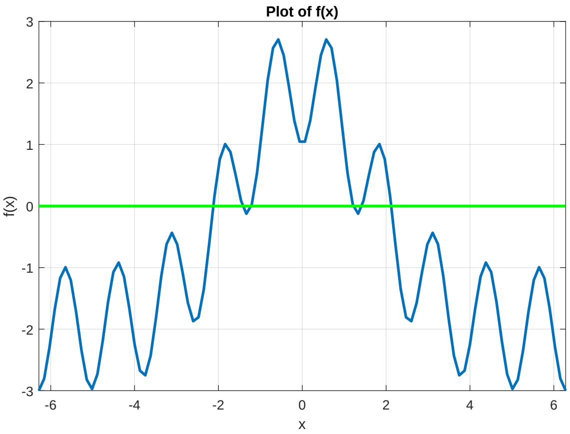
The plot shows that the equation has three solutions.

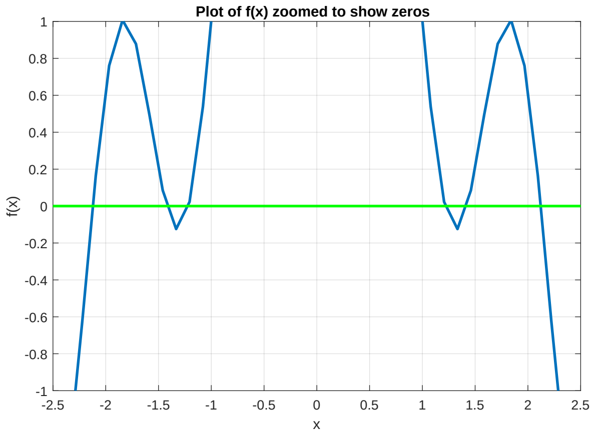
$[a,b]$	f(a)	f(b)	Continuous over $[a,b]$
$[-5,-4]$	0.4942	-0.6162	Yes
$[-1,-0.1]$	0.3194	-2.8238	Yes
$[4,5]$	0.6112	-0.4974	Yes

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

Listing 5: `../src/q1/Q1a_iv_funcPlt.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)	Continuous over $[a, b]$
$[-2.5, -2]$	-1.8518	0.6364	Yes
$[-1.5, -1.25]$	0.2039	-0.0730	Yes
$[-1.25, -1]$	-0.0730	0.9913	Yes
$[1, 1.25]$	0.9913	-0.0730	Yes
$[1.25, 1.5]$	-0.0730	0.2039	Yes
$[2, 2.5]$	0.6364	-1.8518	Yes

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

Listing 6: `../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');

    % intialise 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 $f(x) = x^4 - e^{-x}\cos(x) = 0 \quad x \in [-2\pi, 2\pi]$.

Listing 7: `../src/q1/Q1b_i_funcRoots.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 tolerance to be accurate to 8 significant figures.

$[a,b]$	Root	# Iterations
$[-1.5,-1]$	-1.084359675645828	23
$[0.5,1]$	0.762211065739393	26

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

Listing 8: `../src/q1/Q1b_ii_funcRoots.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
$[-1,-0.5]$	-0.928626310080290	26
$[0.5,1]$	0.928626310080290	26

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

Listing 9: `../src/q1/Q1b_iii_funcRoots.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])
```

Again note the different tolerances due to the different orders of magnitude of the roots.

$[a,b]$	Root	# Iterations
$[-5,-4]$	-4.495372205972672	24
$[-1,-0.1]$	-0.477733755484223	27
$[4,5]$	4.491409689188004	24

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

Listing 10: `../src/q1/Q1b_iv_funcRoots.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
$[-2.5,-2]$	-2.122238188982010	23
$[-1.5,-1.25]$	-1.425543159246445	22
$[-1.25,-1]$	-1.214593321084976	22
$[1,1.25]$	1.214593321084976	22
$[1.25,1.5]$	1.425543159246445	22
$[2,2.5]$	2.122238188982010	23

(c) The iterative scheme we asked to implement is called Steffensen's method[2]. This is implemented in the `steffensenRoot` function.

Listing 11: `../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 interations
    %   err = 1*n vector of the absolute error after each interation
    %
    %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 interation
        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
```

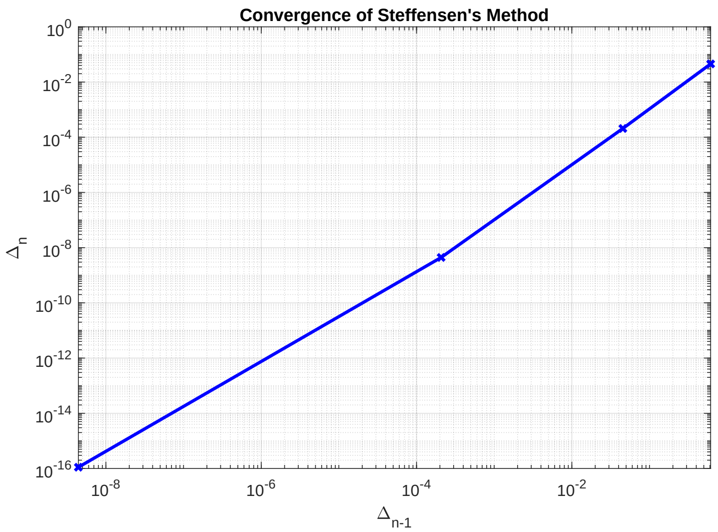
The following uses this function to find the root of $e^{-x} - x = 0$ and calculate the convergence.

Listing 12: `../src/q1/Q1c_errorConvergance.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}');
xlim([e(end - 1) e(1)]);
grid on;
```

```
%% Find order of congerence
polyfit(log(e(1:end-1)), log(e(2:end)), 1)
```

After 5 iterations the root $x = 0.567143290409784$ is accurate to 12 decimal places.



The graph shows a straight which shows 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. The absolute error is an approximate so this can be approximated to 2 meaning Steffensen's method is second order.

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

Listing 13: `../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 iteraterative sequence
    %
    %Usage:
    %   xn = fixedPointRoot(@(x) cos(x), 0.75, 100) -> looks for a
    %   root of the equation x - cos(x) = 0, starting with an inital 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

    % 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
```

Then the `cobwebDiagram` function can display the results of the fixed point iteration.

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

```
function cobwebDiagram(g, x0, n, 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, n);

    %% 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;
    y(y ~= real(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
```

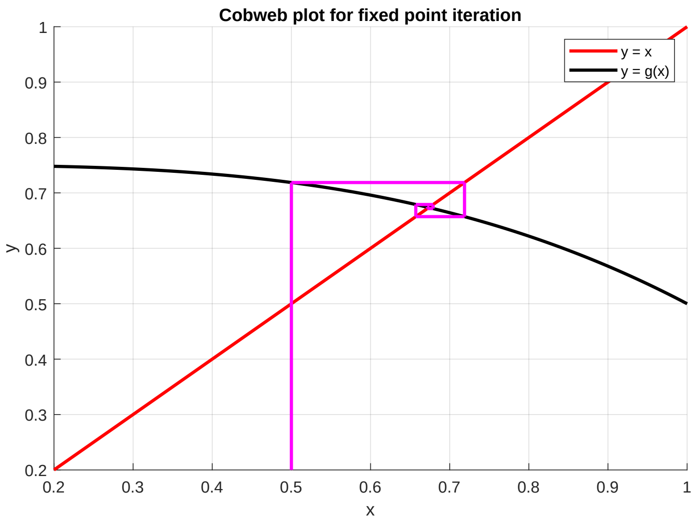
```
        plot([xn(i) xn(i + 1)], [xn(i + 1) xn(i + 1)],'m-');
        plot([xn(i + 1) xn(i + 1)], [xn(i + 1) xn(i + 2)],'m-');
    end
end
```

The function requires user inputs of the iteration function $g(x)$, the initial guess x_0, the number of iterations and the interval to display on the x-axis (which should contain the root). The reason to user has to enter the range to display on the x-axis is because in the case of divergence the estimate of the root can shoot off to very large value which when plotted make previous iteration so small that the plot doesn't show anything meaningful.

The first test I did was to ensure it could produce a cobweb diagram. For this I set $g(x) = \frac{3-x^3}{4}$ with an initial guess of $\frac{1}{2}$.

Listing 15: `../src/q1/Q1d_cobweb_example.m`

```
cobwebDiagram(@(x) (3 - x.^3)/4, 0.5, 10, 0.2, 1)
```

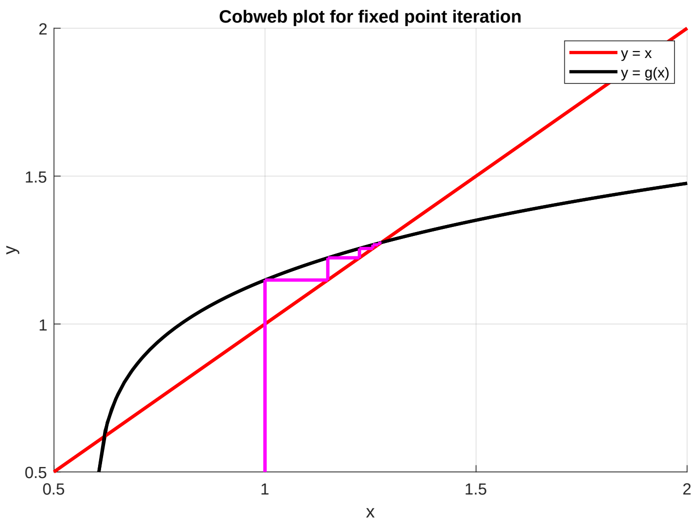


This shows how for this equation fixed point iteration gradually converges by jumping either side of the root. This is because $-1 < g'(x) < 0$ at the root.

Next I tested producing a staircase diagram, so I set $g(x) = \sqrt[5]{5x-3}$ with an initial guess of 1.

Listing 16: `../src/q1/Q1d_staircase_example.m`

```
cobwebDiagram(@(x) (5*x - 3).^0.2, 1, 10, 0.5, 2)
```

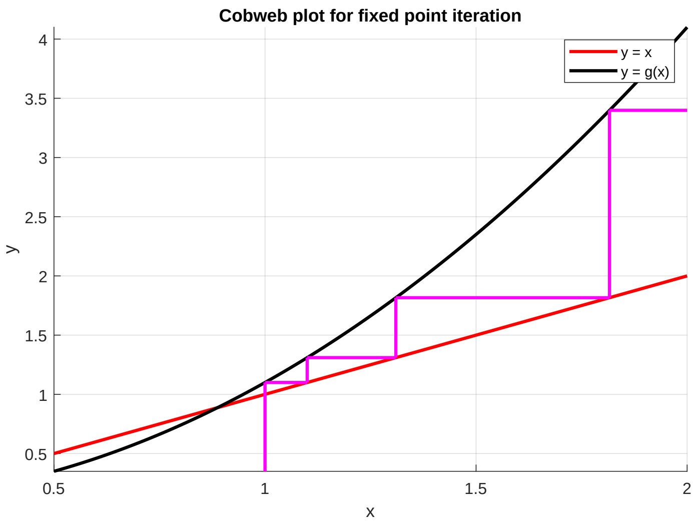


This time fixed point iteration converges nicely to the root in staircase fashion as $0 < g'(x) < 1$ at the root.

Since the function should also be able to handle iterations that don't converge, I tried $g(x) = x^2+0.1$ with an initial guess of 1.

Listing 17: `../src/q1/Q1d_divergance_staircase_example.m`

```
cobwebDiagram(@(x) x.^2 +0.1, 1, 10, 0.5, 2)
```

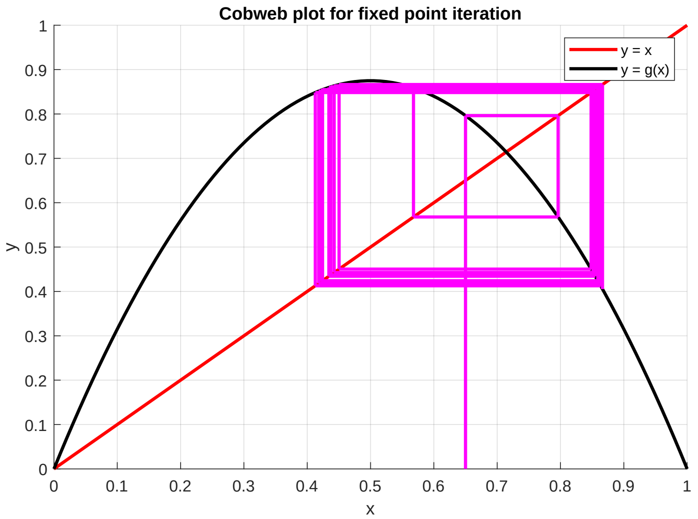


This time the solution diverges away from the root in a staircase fashion as $g'(x) > 0$ at the root.

The function should also handle divergence in a cobweb fashion, so I tried $g(x) = 3.5x(1-x)$ with an intial guess of 0.65.

Listing 18: `../src/q1/Q1d_divergance_cobweb_example.m`

```
cobwebDiagram(@(x) 3.5*x.*(1 - x), 0.65, 20, 0, 1);
```

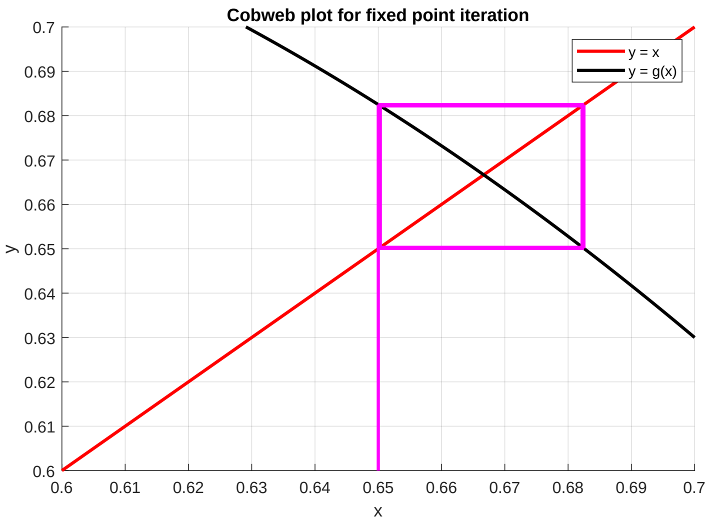


Here the iteration diverges away from the root in a cobweb fashion as $g'(x) < -1$ at the root.

The iteration can also orbit the root. For example when $g(x) = 3x(1 - x)$ with an initial guess of 0.65.

Listing 19: `../src/q1/Q1d_divergance_orbit_example.m`

```
cobwebDiagram(@(x) 3*x.*(1 - x), 0.65, 10, 0.6, 0.7);
```



This is an example of a period 2 orbit.

Question 2: Numerical integration and differentiation

(a) (i) The first of the given expression is Simpson's 3/8 rule and the second given expression is Milne's rule[1]. However, to use the composite version the integral must be broken down into smaller intervals. For example breaking the integral into n intervals gives $\int_a^b f(x)dx = \int_a^{b_1} f(x)dx + \int_{a_1}^{b_2} f(x)dx + \cdots + \int_{a_n}^{b} f(x)dx$. Then each of these smaller integrals can be calculated using either of the methods.

Deriving Composite Simpson's 3/8 rule

The given formula

$$\int_a^b f(x)dx \approx \frac{b-a}{8}\left[f(a) + f(\frac{2a+b}{3}) + f(\frac{a+2b}{3}) + f(b)\right], \tag{1}$$

divides the interval $[a, b]$ into 3 subintervals by creating 4 nodes. The width of the subinterval is $h = \frac{b-a}{3}$ and the position of the node is given by $x_i = a + (i-1)h$ for $i = 1, 2, \ldots, 4$. This means $a = x_1$ and likewise $b = x_4$. So Equation 1 can be rewritten as

$$\int_a^b f(x)dx \approx \frac{3h}{8}\left[f(x_1) + 3f(x_2) + 3f(x_3) + f(x_4)\right]. \tag{2}$$

So to use the composite version the interval $[a, b]$ should be divided into N subintervals where $N \mod 3 \equiv 0$. This will result in subintervals of width $h = \frac{b-a}{N}$ and $N + 1$ nodes positioned at $x_i = a + (i-1)h$ for $i = 1, 2, \ldots, N + 1$. So $a = x_1$ and $b = x_{N+1}$. Doing this gives

$$\begin{aligned}\int_a^b f(x)dx \approx \frac{3h}{8}&\left[f(x_1) + 3f(x_2) + 3f(x_3) + 2f(x_4) + 3f(x_5) + 3f(x_6) + 2f(x_7)\right. \\ &\left. + \cdots + 2f(x_{N-2}) + 3f(x_{N-1}) + 3f(x_N) + f(x_{N+1})\right].\end{aligned} \tag{3}$$

So rewriting as a summation gives

$$\int_a^b f(x)dx \approx \frac{3h}{8}\left[f(x_1) + 3\sum_{i=1}^{N/3}(f(x_{3i-1}) + f(x_{3i})) + 2\sum_{i=1}^{(N/3)-1}(f(x_{3i+1})) + f(x_{N+1})])\right]. \tag{4}$$

Deriving Composite Milne's rule

Similarly, the second given formula

$$\int_a^b f(x)dx \approx \frac{b-a}{3}\left[2f(\frac{3a+b}{4}) - f(\frac{a+b}{2}) + 2f(\frac{a+3b}{4})\right], \tag{5}$$

divides the interval $[a, b]$ into 4 subintervals by creating 5 nodes. The width of the subinterval is $h = \frac{b-a}{4}$ and the position of the node is given by $x_i = a + (i-1)h$ for $i = 1, 2, \ldots, 5$. So $a = x_1$ and $b = x_5$. So Equation 5 can be rewritten as

$$\int_a^b f(x)dx \approx \frac{4h}{3}\left[2f(x_2) - f(x_3) + 2f(x_4)\right]. \tag{6}$$

So to use the composite version the interval $[a, b]$ should be divided into N subintervals where $N \mod 4 \equiv 0$. This will result in subintervals of width $h = \frac{b-a}{N}$ and $N + 1$ nodes positioned at $x_i = a + (i-1)h$ for $i = 1, 2, \ldots, N + 1$. So $a = x_1$ and $b = x_{N+1}$. Doing this gives

$$\begin{aligned}\int_a^b f(x)dx \approx \frac{4h}{3}&\left[2f(x_2) - f(x_3) + 2f(x_4) + 2f(x_6) - f(x_7) + 2f(x_8) + \cdots\right. \\ &\left. + 2(f(x_{N-1})) - f(x_{N-1}) + 2f(x_n)\right].\end{aligned} \tag{7}$$

This can be written as a summation

$$\int_a^b f(x)dx \approx \frac{4h}{3}\left[2\sum_{i=1}^{N/2}(f(x_{2i})) - \sum_{i=1}^{N/4}(f(x_{4i-1}))\right] \tag{8}$$

So Equation 4 and Equation 8 can be implemented in MATLAB to give the composite version of the fomula provided.

Listing 20: ../src/q2/compositeQuad.m

```
function [compQuad, h, err] = compositeQuad(f, mode, a, b, tol)
   %compositeQuad amproximate quadrature using a Newton-Coutes method
   %
   %Inputs:
   %   f = function handle of the integrand
   %   mode = the newton-coutes method to use: 1 - Simpson's 3/8, 2 -
   %   Milne's rule
   %   a = lower bound of the interval
   %   b = upper bound of the interval
   %   tol = desired absolute error tolerance
   %
   %Outputs:
   %   compQuad = vector of the sucessive 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), 1, 1, 5e-4) -> Estimates the quadrature
   %   of e^x in the interval [0,1] using the Simpson's 3/8 rule
   %   to 3 decimal places

   if mode == 1
      % Simpson's 3/8 rule
      % number of subintervals must be a multiple of 3
      N = 3;
      q = @(x, h) (3*h)/8*(f(x(1)) + 3*sum(f(x(2:3:end-2))...
          + f(x(3:3:end-1))) + 2*sum(f(x(4:3:end-3))) + f(x(end)));
   elseif mode == 2
      % Milne's rule
      % number of subintervals must be a multiple of 4
      N = 4;
      q = @(x, h) (4*h)/3*(2*sum(f(x(2:2:end-1))) - sum(f(x(3:4:end-2))));
   else
     error("Invalid mode")
   end

   % max number of iterations to prevent infinite loop
   nMax = 50;
   % iteration counter
   n = 1;
   % 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;

      % generate equally spaced node
      x = linspace(a, b, N + 1);
      % calculate quadrature
      compQuad(n) = q(x, h(n));
```

```
        try
            err(n) = abs(compQuad(n) - compQuad(n - 1));
        catch
            % prevents error when calculating first error term as no
            % previous approximation to compare againsy
            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 $\int_0^5 e^x - xdx$ to 6 decimal places as follows.

Listing 21: `../src/q2/Q2ai_exampleIntegration.m`

```
% intergrand
f = @(x) exp(x) - x;
% interval
a = 0;
b = 5;
tol = 5e-7;
% using Simpson's 3/8 rule
compositeQuad(f, 1, a, b, tol)
% using Milne's rule
compositeQuad(f, 2, a, b, tol)
```

Both give the answer to the example as 134.913159.

(ii) The order of a Newton-Cotes method is measured with respect to h (the size of subintervals).

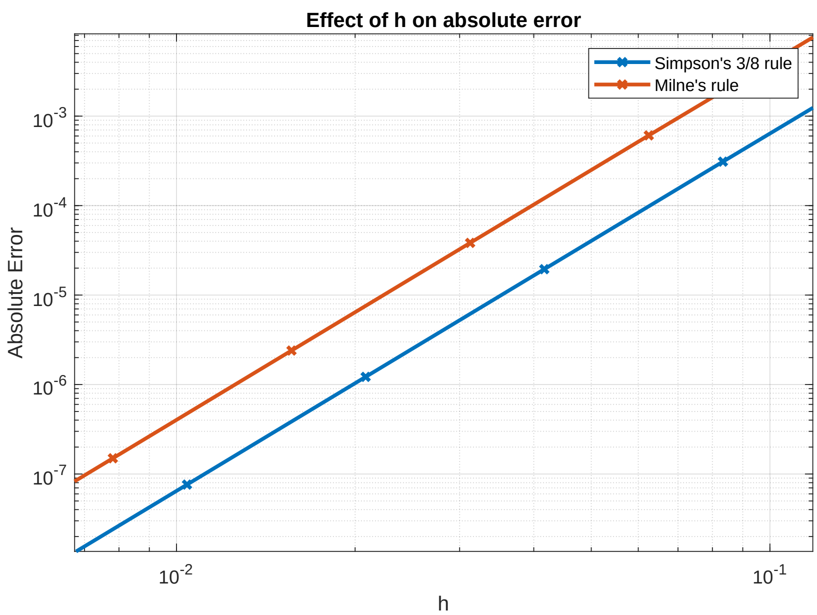


Figure 1: Error convergance of composite Simpson's 3/8 rule and composite Milne's rule. Generated using Listing 31.

Figure 1 shows both method produce a straight line which shows error $\propto h^q$ where q is the gradient. The `polyfits` function shows both line have a gradient of 4 which means both composite Simpson's 3/8 rule and composite Milne's rule are 4th order.

The accuracy is the highest order polynomial that the method can integrate exactly. So to find the degree of accuracy of each method test it on a variety of polynomials of increasing order and see if it can integrate the polynomial exactly.

	Exact Value	Simpon's 3/8 Rule	Milne's Rule
$\int_0^1 x\ dx$	$\frac{1}{2}$	0.5	0.5
$\int_0^1 x^2\ dx$	$\frac{1}{3}$	0.333333333333333	0.333333333333333
$\int_0^1 x^3\ dx$	$\frac{1}{4}$	0.25	0.25
$\int_0^1 x^4\ dx$	$\frac{1}{5}$	0.203703703703704	0.192708333333333

This shows that both methods are of degree of accuracy 3 as they can both exact integrate cubics but not quartics.

Whilst both methods are fourth order Simpson's 3/8 rule has a lower error term so is slightly better as it is more accurate. This is shown in Figure 1 as the absolute error for Simpson's 3/8 rule is always less.

(b) (i) To find the order of the rounding and truncation error when numerically approximating $f''(x)$ the absolute error is plotted against h. Since $f(x)$ given is differentiable $f''(x)$ can be found exactly and then the absolute error can be found by subtracting Equation 10 from the numerical approximation. The exact second derivative is

$$\frac{d^2}{dx^2}(\sin^3(x)) = -3\sin^3(x) + 3\cos(x)\sin(2x). \tag{9}$$

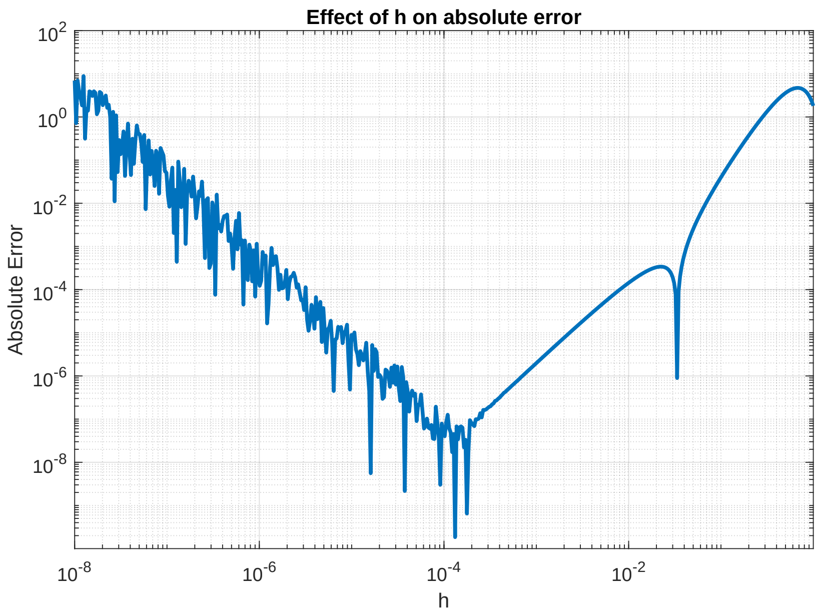


Figure 2: Absolute error when numerically approximating the second derivative. Plot generated by Listing 32

The gradient of the two sections of the graph shows the order of the rounding and truncation error. The rounding error is shown by the jagged first half which has a gradient of -2 which shows the order of the rounding error is $\mathcal{O}(h^{-2})$. The second half of the graph shows the truncation error. This has a gradient of 2 which shows the truncation error is $\mathcal{O}(h^2)$.

(ii) The error comprises the truncation and rounding error. To find the truncation error of

$$f''(x) \approx \frac{2f(x) - 5f(x+h) + 4f(x+2h) - f(x+3h)}{h^2}, \tag{10}$$

consider the following Taylor expansions

$$f(x+h) = f(x) + hf'(x) + \frac{h^2}{2}f''(x) + \frac{h^3}{6}f^{(3)}(x) + \frac{h^4}{24}f^{(4)}(x) + \cdots, \tag{11}$$

$$f(x+2h) = f(x) + 2hf'(x) + \frac{4h^2}{2}f''(x) + \frac{8h^3}{6}f^{(3)}(x) + \frac{16h^4}{24}f^{(4)}(x) + \cdots, \tag{12}$$

$$f(x+3h) = f(x) + 3hf'(x) + \frac{9h^2}{2}f''(x) + \frac{27h^3}{6}f^{(3)}(x) + \frac{81h^4}{24}f^{(4)}(x) + \cdots. \tag{13}$$

Substituting (11), (12), (13) into (10) gives

$$f''(x) \approx \frac{h^2 f''(x) + \frac{-11}{12}h^4 f^{(4)}(x) + \mathcal{O}(h^5)}{h^2} \tag{14}$$

$$= f''(x) - \frac{11}{12}h^2 f^{(4)}(x) + \mathcal{O}(h^3), \tag{15}$$

so the truncation error is $-\frac{11}{12}h^2 f^{(4)}(x) + \mathcal{O}(h^3)$.

To find the rounding error assume h is small and can be stored exactly. So the rounding error in storing $f(x)$, $f(x+h)$, $f(x+2h)$ and $f(x+3h)$ is $|f(x)|\epsilon$, where ϵ is the floating point relative accuracy, 2.2204×10^{-16}. This means the total rounding error is $\frac{12|f(x)|\epsilon}{h^2}$. So the total error is given by

$$\text{error} \approx \frac{11}{12}h^2|f^{(4)}(x)| + \frac{12|f(x)|\epsilon}{h^2}. \tag{16}$$

We want to minimise the error so

$$\frac{d}{dt}\text{error} \approx \frac{22}{12}h|f^{(4)}(x)| - \frac{24|f(x)|\epsilon}{h^3} = 0. \tag{17}$$

If we didn't know what $f^{(4)}(x)$ was we could approximate $f(x) \approx f^{(4)}(x)$. However in this example it is possible to find $f^{(4)}(x)$ exactly

$$\frac{d^4}{dx^4} = -24\cos(x)\sin(2x) + 9\sin^3(x) - 12\sin(x)\cos(2x). \tag{18}$$

Solving Equation 17 to find the h that minimises the absolute error when finding the numerical second derivative at x gives

$$h^\star = \sqrt[4]{\frac{144|f(x)|}{11|f^{(4)}(x)|}\epsilon}. \tag{19}$$

When $x = 1$ then $h^\star \approx 1.67 \times 10^{-4}$.

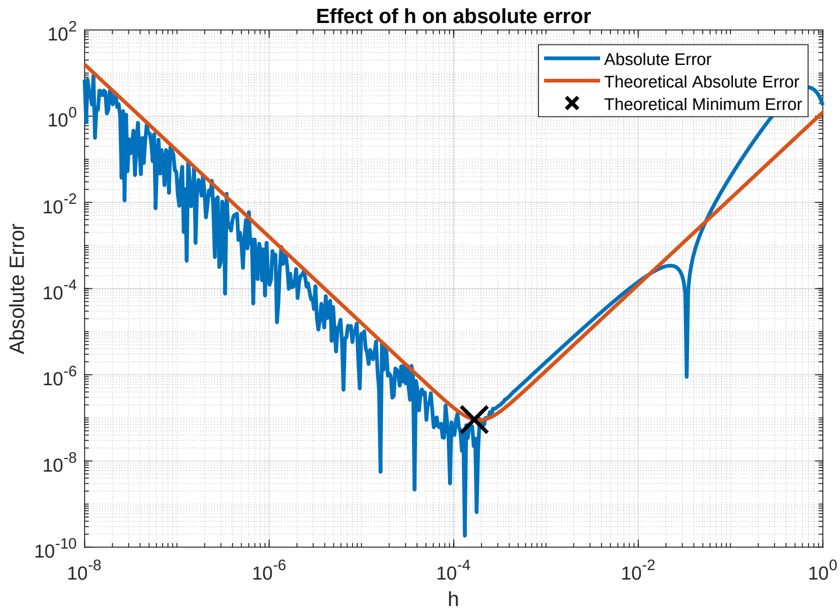


Figure 3: Plot produced using Listing 33

The algebraic expression has the same gradient however it is slightly offset. This is because it assumes that h can be stored exactly and because some higher order terms that would have a small impact are ignored. However, the value of h that minimises error lines up correctly.

Question 3: Numerical solution of ODEs

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

(a)
```
function dydt = rhsProjectile(t, y, g, mu)
    %rhsProjectile returns colomn 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 23: ../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
    %   varibles at the corisponding value of t in the same fashion as
```

```
    %   ode45
    %Usage:
    %   [t,y]=forwardEulerProjectile(@rhsProjectile, [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 matix
    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 24: `../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
    %   varibles at the corisponding value of t in the same fashion as
    %   ode45
    %Usage:
    %   [t,y]=rk4Projectile(@rhsProjectile, [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 matix
    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`

Listing 25: `../src/q3/ode45projectile.m`

```
[t, y] = ode45(@(t, y) rhsProjectile(t, y, g, mu), tSpan, y0');
```

(c) Solving the ODE using the three solvers gives the following different trajectories.

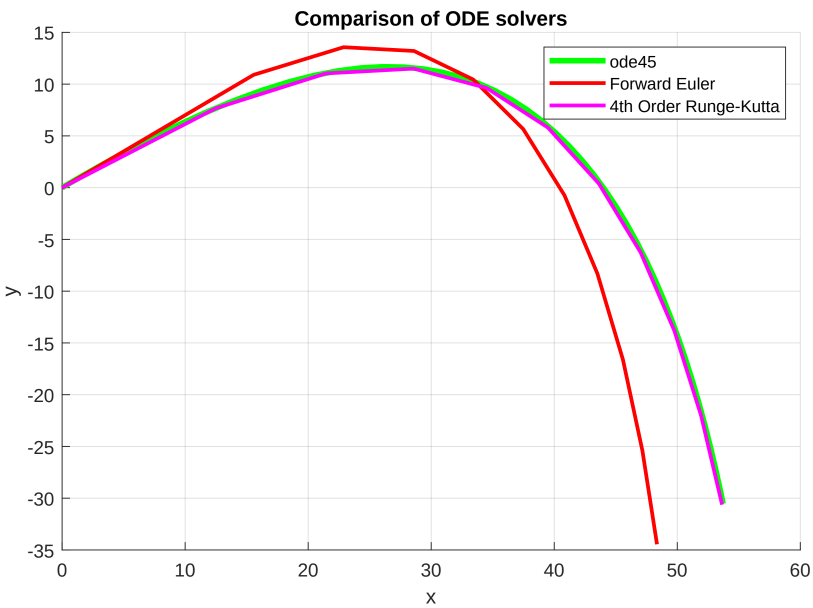


Figure 4: Comparision of the three different solutions. Plot produced using Listing 34

(d) Global truncation error of forward Euler compared to `ode45`.

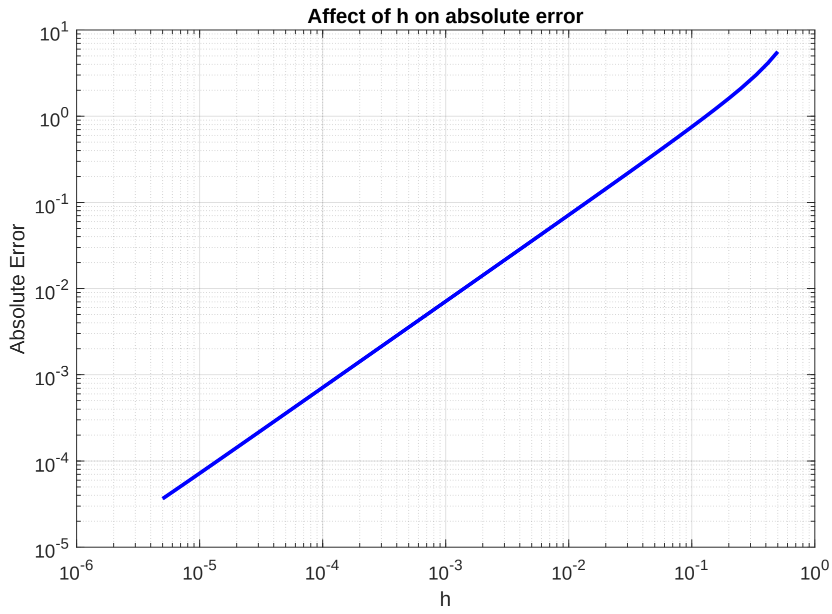


Figure 5: Generated using Listing 35

The order of global truncation error can be given by the gradient in the limit as $h \to 0$. In this case the gradient is 1 so the global truncation error is $\mathcal{O}(h)$.

(e) To find when the projectile crosses the x-axis first an event function is created.

Listing 26: ../src/q3/xaxisEvent.m

```
function [value, isTerminal, direction] = xaxisEvent(t, y)
    % halt when the ball reaches xaxis ie. y = 0
    value = y(2);
    % end integration
    isTerminal = 1;
    % ball should be falling
    direction = -1;
end
```

Then solve the ODE with the added event function.

Listing 27: ../src/q3/Q3e_eventDetection.m

```
%% setup parameters
tSpan = [0 5];
v0 = 38;
theta0 = deg2rad(35);
g = 9.81;
mu = 2.79e-2;
h = 0.5;
n = (tSpan(end) - tSpan(1))/h;

%initial conditions
y0 = [0 0 v0*cos(theta0) v0*sin(theta0)]';

%% solve ODE
options = odeset("Events", @xaxisEvent);
[t, y, te, ye , ie] = ode45(@(t, y) rhsProjectile(t, y, g, mu), tSpan,...
    y0, options);
```

```
%% display result
fprintf("Projectile first crosses x axis at %.13f\n", te)
```

This gives that the projectile first passes through the x-axis when $t = 3.0387773687184$ and $x = 43.990576824668466$m.

(f) An equivalent formulation to this problem is to consider the angles θ_0 required so that when a particle is fired from $(-40, 0)$ it lands through the origin $(0, 0)$. This can be visualized by plotting the landing x coordinate against θ_0 as shown in Figure 6.

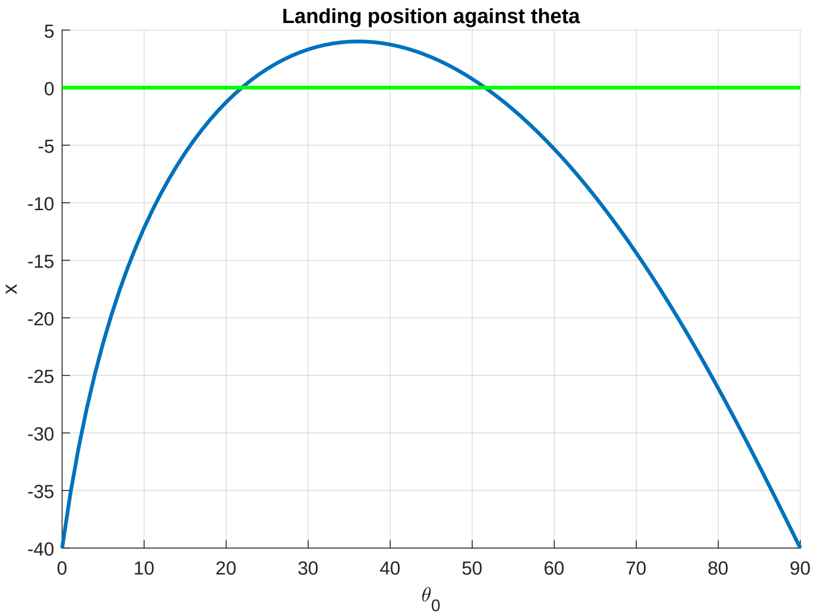


Figure 6: Generated using Listing 36

This shows the problem can be reduced to a root finding problem. This can be done using the bisection method already implemented in Listing 6 or Stephen's method implemented in Listing 11. The two values of θ_0 can be bracketed by the intervals $[20, 25]$ and $[50, 55]$. First a function, $x(\theta_0)$, that returns just the landing position given a θ_0 needs to be created.

Listing 28: `../src/q3/landingPosition.m`

```
function x = landingPosition(theta)
    %% setup parameters
    tSpan = [0 5];
    v0 = 38;
    g = 9.81;
    mu = 2.79e-2;
    theta0 = deg2rad(theta);

    %% solve ODE for a range of theta
    %set event detection
    options = odeset("Events", @xaxisEvent);
    % calculate initial conditions for a given theta
    y0 = [-40 0 v0*cos(theta0) v0*sin(theta0)]';
    % solve ODE
    [t, y, te, ye , ie] = ode45(@(t, y) rhsProjectile(t, y, g, mu), tSpan,...
```

```
    y0, options);

    x = ye(1);
end
```

Then using the bisection method the roots of $x(\theta_0) = 0$ can be found.

Listing 29: `../src/q3/Q3f_thetaSolution.m`

```
bisectRoot(@landingPosition, 20,25, 5e-20)
bisectRoot(@landingPosition, 50,55, 5e-20)
```

This gives $\theta_0 = 21.9185709$ or $\theta_0 = 51.567361$ as solutions to the initial angle so that the projectile crosses the x-axis 40m away.

References

[1] Wikipedia. *Newton–Cotes formulas — Wikipedia, The Free Encyclopedia*. 2021. URL: `https://en.wikipedia.org/wiki/Newton%E2%80%93Cotes_formulas` (visited on 01/13/2021).

[2] Wikipedia. *Steffensen's method — Wikipedia, The Free Encyclopedia*. 2021. URL: `https://en.wikipedia.org/wiki/Steffensen%27s_method` (visited on 01/13/2021).

Appendix A Additional Code for Question 1

Listing 30: ../src/q1/pltFunc.m

```
function pltFunc(f, domain, discontLim)
    %pltFunc plots function f between values of xLim removing any values
    % that are greater than discontLim to prevent MATLAB plotting
    % discontinuous functions as continous 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
```

Appendix B Additional Code for Question 2

Listing 31: ../src/q2/Q2aii_errorConvergance.m

```
% integrand
f = @(x) exp(x) - x;
% interval
a = 3;
b = 4;
tol = 5e-9;
% integrate using Simpson's 3/8 rule
[q, hSimp38, errSimp38] = compositeQuad(f, 1, a, b, tol);
% integrate using Milne's rule
[q, hMilne, errMilne] = compositeQuad(f, 2, a, b, tol);

% plot error
loglog(hSimp38(2:end), errSimp38,"x-",...
```

```
    hMilne(2:end), errMilne,"x-", "LineWidth",2);
legend("Simpson's 3/8 rule", "Milne's rule")
xlim([hSimp38(end) hSimp38(2)]);
xlabel("h")
ylabel("Absolute Error")
title("Effect of h on absolute error")
grid on;

% find gradient
polyfit(log(hSimp38(2:end)), log(errSimp38), 1)
polyfit(log(hMilne(2:end)), log(errMilne), 1)
```

Listing 32: `../src/q2/Q2bi_absoluteError.m`

```
f = @(x) sin(x).^3;
% numerical second derivative
d2 = @(f, x, h) (2*f(x) - 5*f(x + h) + 4*f(x + 2*h) - f(x + 3*h))./(h.^2);
% true second derivative
d2True = @(x) -3*sin(x)^3 + 3*cos(x)*sin(2*x);

% x value to find derivative at
x = 1;

h = logspace(-8,0,500);

% calculate vector of second derivatives
df2 = d2(f, x, h);
% calculate absolute error
err = abs(df2 - d2True(x));

% plot absolute error
loglog(h,err,"LineWidth",2);
grid on
xlabel("h")
ylabel("Absolute Error")
title("Effect of h on absolute error")
```

Listing 33: `../src/q2/Q2bii_absoluteErrorEstimate.m`

```
f = @(x) sin(x).^3;
% numerical second derivative
d2 = @(f, x, h) (2*f(x) - 5*f(x + h) + 4*f(x + 2*h) - f(x + 3*h))./(h.^2);
% true second derivative
d2True = @(x) -3*sin(x)^3 + 3*cos(x)*sin(2*x);
% true fourth derivative
d4True = @(x) -24*cos(x)*sin(2*x) + 9*sin(x)^3 - 12*sin(x)*cos(2*x);

% x value to find derivative at
x = 1;

h = logspace(-8,0,500);

df2 = d2(f, x, h);
err = abs(df2 - d2True(x));
errTheoretical = @(h) (11/12)*h.^2*f(x)*2.26 + (12*f(x)*eps)./(h.^2);
hstar = nthroot((144*eps*f(x))/(11*abs(d4True(x))), 4);
estar = errTheoretical(hstar);

loglog(h,err,h, errTheoretical(h),"LineWidth",2);

hold on
```

```
plot(hstar,estar,"kx","MarkerSize",20, "LineWidth",2);

legend("Absolute Error", "Theoretical Absolute Error",...
   "Theoretical Minimum Error")
grid on
xlabel("h")
ylabel("Absolute Error")
title("Effect of h on absolute error")
```

Appendix C Additional Code for Question 3

Listing 34: `../src/q3/Q3c_trajectoryComparison.m`

```
%% setup parameters
tSpan = [0 5];
v0 = 38;
theta0 = deg2rad(35);
g = 9.81;
mu = 2.79e-2;
h = 0.5;
n = (tSpan(end) - tSpan(1))/h;

%initial conditions
y0 = [0 0 v0*cos(theta0) v0*sin(theta0)]';

%% solve ODE
%ode45
[t, yode45] = ode45(@(t, y) rhsProjectile(t, y, g, mu), tSpan, y0);

%forward euler
[t, yFE] = forwardEulerProjectile(@rhsProjectile, tSpan, y0, g, mu, n);

%runge-kutta 4th order
[t, yRK4] = rk4Projectile(@rhsProjectile, tSpan, y0, g, mu, n);

%% plot figure
% configure plot
hold on
grid on
legend
set(gca, "DefaultLineLineWidth", 2);
xlabel("x")
ylabel("y")
title("Comparison of ODE solvers")

% plot data
plot(yode45(:,1), yode45(:,2), "g-", "DisplayName", "ode45",...
   "LineWidth", 3)
plot(yFE(:,1), yFE(:,2), "r-", "DisplayName", "Forward Euler")
plot(yRK4(:,1), yRK4(:,2), "m-", "DisplayName", "4th Order Runge-Kutta")
```

Listing 35: `../src/q3/Q3d_truncationError.m`

```
%% setup parameters
tSpan = [0 5];
v0 = 38;
theta0 = deg2rad(35);
g = 9.81;
mu = 2.79e-2;
%number of steps
```

```
n = floor(logspace(1,6));
%preallocate step size and error
h = nan(size(n));
err = nan(size(n));

%initial conditions
y0 = [0 0 v0*cos(theta0) v0*sin(theta0)]';

%% solve ODE
%use ode45 to get "exact" solution
options = odeset("RelTol",1e-6,"AbsTol",1e-9);
sol = ode45(@(t, y) rhsProjectile(t, y, g, mu), tSpan, y0, options);

for i = 1:length(n)
    %find the solution using FE for the given number of steps
    [t, yFE] = forwardEulerProjectile(@rhsProjectile, tSpan, y0,...
       g, mu, n(i));
    %interpolate ode45 at the correct times
    yExact = deval(sol, t)';

    %calculate step size
    h(i) = (t(end) - t(1))/n(i);
    %find the local error at each step
    localErr = vecnorm(yFE - yExact, 2, 2);
    %integrate to find global error
    err(i) = trapz(t,localErr)/t(end);
end

%% Plot error convergance
loglog(h, err,"b-","LineWidth",2)
ylabel("Absolute Error")
xlabel("h")
title("Affect of h on absolute error")
grid on
polyfit(log(h), log(err),1)
```

Listing 36: `../src/q3/Q3f_angleTrajectory.m`

```
%% setup parameters
tSpan = [0 5];
v0 = 38;
g = 9.81;
mu = 2.79e-2;

%% solve ODE for a range of theta
%set event detection
options = odeset("Events", @xaxisEvent);

% generate range of theta
theta = [0:90];
landingPosition = zeros(1,91);

for i=theta
    theta0 = deg2rad(i);
    % calculate initial conditions for a given theta
    y0 = [-40 0 v0*cos(theta0) v0*sin(theta0)]';
    % solve ODE
    [t, y, te, ye , ie] = ode45(@(t, y) rhsProjectile(t, y, g, mu), tSpan,...
    y0, options);
    % extract landing position
```

```
    landingPosition(i+1) = ye(1);
end

%% Plot results
% configure plot
hold on
grid on
set(gca, "DefaultLineLineWidth", 2);
xlabel("\theta_{0}")
ylabel("x")
title("Landing position against theta")

plot(theta,landingPosition)
plot([0 90], [0 0],'g-')
```