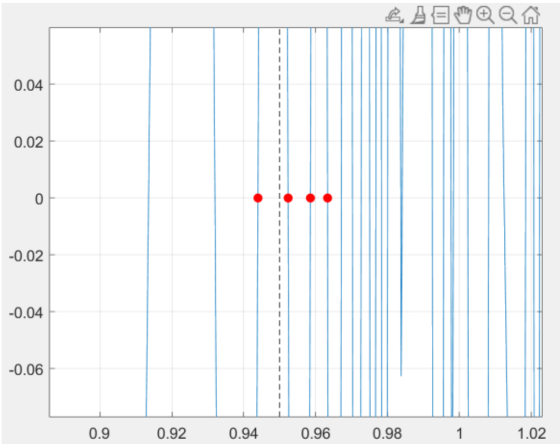
Assignment 1

111550142 尤瑋辰

1. I use MATLAB to calculate the result. Here is the result and code:




```
Four zeros nearest to x = 0.95:
    0.9524
    0.9440
    0.9586
    0.9633

Average of iterations
    13
```

```
% Define the range near x = 0.95 and Number of intervals to check
x_min = 0.9;
x_max = 1;
num_intervals = 1000;

% Initialize an array to store the intervals
intervals = zeros(num_intervals, 2);    % num_intervals * 2 matrix

% Find intervals where the function changes sign
count = 0;
for i = 1:num_intervals
    x1 = x_min + (x_max - x_min) * (i - 1) / num_intervals;
    x2 = x_min + (x_max - x_min) * i / num_intervals;
    if sign(f(x1)) ~= sign(f(x2))
        count = count + 1;
        intervals(count, :) = [x1, x2];
    end
end

% Discard unused rows
intervals = intervals(1:count, :);

% Use these intervals as starting intervals for bisection method
% Perform bisection on each interval to find the zeros
tol = 1e-8; % Tolerance
max_iter = 100; % Maximum number of iterations

roots = [];
iters = [];
for i = 1:size(intervals, 1)
    [root, iter] = bisection(f, intervals(i, 1), intervals(i, 2), tol, max_iter);
    roots = [roots; root];  % Add new root to roots[]
    iters = [iters; iter];
end
```

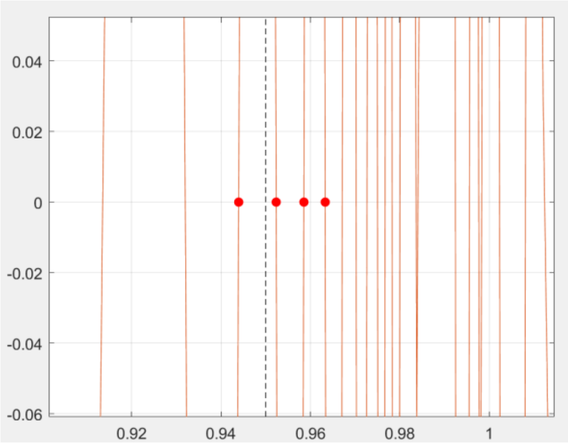
```
function [root, iter]= bisection(f, a, b, tol, max_iter)
    if sign(f(a)) == sign(f(b))
        error(['Function has same signs at both ends of the interval.'
            ' Bisection method cannot guarantee convergence.'])
    end

    iter = 0;
    while (b - a) / 2 > tol && iter < max_iter
        c = (a + b) / 2;
        if f(c) == 0
            root = c;
            return;
        elseif(f(c) * f(a) >= 0) % sign(f(c)) == sign(f(a))
            a = c;
        else
            b = c;
        end
        iter = iter + 1;
    end
    root = (a + b) / 2;
end
```

Explanation:

1. Divide the range (0.9 ~ 1) to 1000 intervals.
2. For each interval, check whether the function changes sign in this interval.
3. For each interval where the function changes sign, use bisection method to find the root.

2. Same as (1), but this time change bisection to secant. Here is the result and code:




```
Four zeros nearest to x = 0.95:
     0.9524
     0.9440
     0.9586
     0.9633

Average of iterations
     5.4074
```

```
% Define the range near x = 0.95 and Number of intervals to check
x_min = 0.9;
x_max = 1;
num_intervals = 1000;

% Initialize an array to store the intervals
intervals = zeros(num_intervals, 2);    % num_intervals * 2 matrix

% Find intervals where the function changes sign
count = 0;
for i = 1:num_intervals
    x1 = x_min + (x_max - x_min) * (i - 1) / num_intervals;
    x2 = x_min + (x_max - x_min) * i / num_intervals;
    if sign(f(x1)) ~= sign(f(x2))
        count = count + 1;
        intervals(count, :) = [x1, x2];
    end
end

% Discard unused rows
intervals = intervals(1:count, :);

% Use these intervals as starting intervals for bisection method
% Perform bisection on each interval to find the zeros
tol = 1e-8; % Tolerance
max_iter = 100; % Maximum number of iterations

roots = [];
iters = [];
for i = 1:size(intervals, 1)
    [root, iter] = secant(f, intervals(i, 1), intervals(i, 2), tol, max_iter);
    roots = [roots; root];  % Add new root to roots[]
    iters = [iters; iter];
end
```

```
function [x2, iter] = secant(f, x0, x1, tol, max_iter)

if abs(f(x0)) < abs(f(x1))
        temp = x0;
        x0 = x1;
        x1 = temp;
end

iter = 0;
err = tol + 1;
while err > tol && iter < max_iter
    x2 = x1 - f(x1)*(x0 - x1)/(f(x0) - f(x1));

    err = abs(x2 - x1);
    x0 = x1;
    x1 = x2;
    iter = iter + 1;
end
```

Explanation:

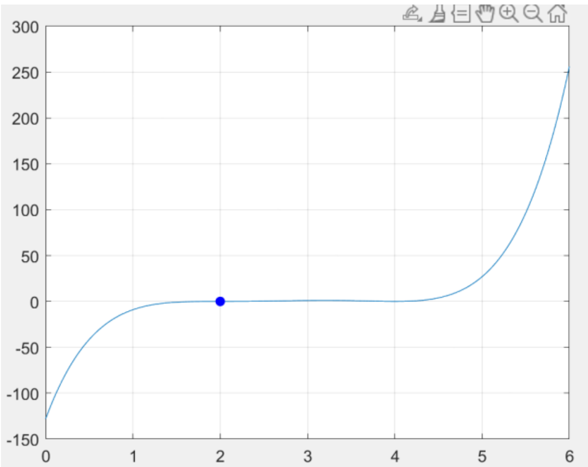
1. Divide the range (0.9 ~ 1) to 1000 intervals.
2. For each interval, check whether the function changes sign in this interval.
3. For each interval where the function changes sign, use secant method to find the root.

The iterations of secant method (avg: 5.4074) are much fewer than those of bisection methods (avg: 13).

3. (a) Root 2 can be found by bisection method, but 4 can't. Because 4 is a double root.

(b) Root 2 can be found by secant method, but 4 can't. Because 4 is a double root.

(c) All of this method get root 2. Here is result and code:




```
>> Q3
bisection method:
	2
secant method:
	2
false position method:
	2
```

```matlab
% Define the function
f = @(x) x.^5 - 14*x.^4 + 76*x.^3 - 200*x.^2 + 256 * x - 128;

% Plot the formula
X = linspace(0, 6, 2000);
plot(X, f(X));
hold on
grid on

x0 = 1;
x1 = 5;
tol = 1e-5;
max_iter = 100;

root1 = bisection(f, x0, x1, tol, max_iter);
root2 = secant(f, x0, x1, tol, max_iter);
root3 = false_position(f, x0, x1, tol, max_iter);

fprintf("bisection method:\n\t%d\n", root1);
fprintf("secant method:\n\t%d\n", root2);
fprintf("false position method:\n\t%d\n", root3);

scatter(root1, f(root1), 'red', 'filled');
scatter(root2, f(root2), 'green', 'filled');
scatter(root3, f(root3), 'blue', 'filled');

hold off
```

```matlab
function [x2, iter] = false_position(f, x0, x1, tol, max_iter)

x2 = x1 - f(x1)*(x0 - x1)/(f(x0) - f(x1));

iter = 0;
while abs(f(x2)) > tol && iter < max_iter
   x2 = x1 - f(x1)*(x0 - x1)/(f(x0) - f(x1));

   if(f(x2) * f(x0) < 0)
      x1 = x2;
   else
      x0 = x2;
   end
   iter = iter + 1;
end
```

4. I use MATLAB to calculate the result. Here is the result and code:

```
>> Q4
Root near 0.6 of the function in a: 6.058296e-01
Root near   1 of the function in b: 1.241143e+00
Root near  -2 of the function in b: -2.211438e+00
```

```matlab
% Define the function
f = @(x) 4*x.^3 - 3*x.^2 + 2*x - 1;

% Initial guesses near x = 0.6
x0 = 0.1;
x1 = 0.6;
x2 = 1.1;

% Tolerance
tol = 1e-8;

% Maximum number of iterations
max_iter = 100;

% Call Muller's method
root = muller(f, x0, x1, x2, tol, max_iter);

% Display the root
fprintf('Root near 0.6 of the function in a:
```

```matlab
% Define the function
f = @(x) x.^2 + exp(x) - 5;

% Initial guesses near x = 1
x0 = 0.5;
x1 = 1;
x2 = 1.5;

% Tolerance
tol = 1e-8;

% Maximum number of iterations
max_iter = 100;

% Call Muller's method
root = muller(f, x0, x1, x2, tol, max_iter);

% Display the root
fprintf('Root near   1 of the function in b:
```

```matlab
% Define the function
f = @(x) x.^2 + exp(x) - 5;

% Initial guesses near x = -2
x0 = -3;
x1 = -2;
x2 = -1;

% Tolerance
tol = 1e-8;

% Maximum number of iterations
max_iter = 100;

% Call Muller's method
root = muller(f, x0, x1, x2, tol, max_iter);

% Display the root
fprintf('Root near  -2 of the function in b:
```

```matlab
function root = muller(f, x0, x1, x2, tol, max_iter)
    iter = 0;
    while iter < max_iter
        h1 = x1 - x0;
        h2 = x2 - x1;
        d1 = (f(x1) - f(x0)) / h1;
        d2 = (f(x2) - f(x1)) / h2;

        a = (d2 - d1) / (h2 + h1);
        b = a * h2 + d2;
        c = f(x2);
        D = sqrt(b^2 - 4 * a * c);

        if abs(b + D) > abs(b - D)
            E = b + D;
        else
            E = b - D;
        end

        dxr = -2 * c / E;
        x3 = x2 + dxr;

        if abs(dxr) < tol * max(abs(x2), 1)
            root = x3;
            return;
        end

        x0 = x1;
        x1 = x2;
        x2 = x3;

        iter = iter + 1;
    end
    root = x3;
end
```

FIGURE 7.4
Pseudocode for Müller's method.

```
SUB Muller(xr, h, eps, maxit)
x2 = xr
x1 = xr + h*xr
x0 = xr − h*xr
DO
  iter = iter + 1
  h0 = x1 − x0
  h1 = x2 − x1
  d0 = (f(x1) − f(x0)) / h0
  d1 = (f(x2) − f(x1)) / h1
  a = (d1 − d0) / (h1 + h0)
  b = a*h1 + d1
  c = f(x2)
  rad = SQRT(b*b − 4*a*c)
  If |b+rad| > |b−rad| THEN
    den = b + rad
  ELSE
    den = b − rad
  END IF
  dxr = −2*c / den
  xr = x2 + dxr
  PRINT iter, xr
  IF (|dxr| < eps*xr OR iter >= maxit) EXIT
  x0 = x1
  x1 = x2
  x2 = xr
END DO
END Müller
```

I refer the pseudocode of this materials: https://reurl.cc/bDv3Zo.

5. (a) I use MATLAB to calculate the result. Here is the result and code:

```
>> Q5
Approximate root x0 = 1 postitive value used: 1.487962e+00
Approximate root x0 = 1 negative value used:: -5.398353e-01
```

```matlab
%f = @(x) exp(x) - 2*x^2;
g_positive = @(x) sqrt(exp(x)/2);
g_negative = @(x) -sqrt(exp(x)/2);

% Set initial guess
x0 = 1; % can change

% Tolerance
tol = 1e-8;

% Maximum number of iterations
max_iter = 100;

% Perform fixed-point iteration for positive root
root_positive = fixed_point(g_positive, x0, tol, max_iter);
fprintf('Approximate root x0 = %.01d postitive value used: %d\n', x0, root_positive);

% Perform fixed-point iteration for negative root
root_negative = fixed_point(g_negative, x0, tol, max_iter);
fprintf('Approximate root x0 = %.01d negative value used:: %d\n', x0, root_negative);
```

```matlab
function root = fixed_point(g, x0, tol, max_iter)
    x = x0;
    iter = 0;
    while iter < max_iter
        x_next = g(x);
        if abs(x_next - x) < tol
            root = x_next;
            return;
        end
        x = x_next;
        iter = iter + 1;
    end
    root = x;
end
```

(b) Use the same code as in part (a), but adjust the initial guess x0.

```
>> Q5
Approximate root x0 = 2.5e+00 postitive value used: 1.487962e+00
Approximate root x0 = 2.5e+00 negative value used:: -5.398353e-01
```

```
>> Q5
Approximate root x0 = 2.7e+00 postitive value used: Inf
Approximate root x0 = 2.7e+00 negative value used:: -5.398353e-01
```

(c) By following the derivations, we obtain the function h(x) and implement it in the code:

$$f(x) = e^x - 2x^2 = 0$$

$$e^x = 2x^2$$

$$x = \log(2x^2)$$

$$h(x) = \log(2x^2)$$

```
%f = @(x) exp(x) - 2*x^2;
h = @(x) log(2*x.^2);

% Set initial guess
x0 = 2.7; % can change

% Tolerance
tol = 1e-8;

% Maximum number of iterations
max_iter = 100;

% Perform fixed-point iteration for positive root
root = fixed_point(h, x0, tol, max_iter);
fprintf('Approximate root x0 = %.01d h(x) used: %d\n', x0, root);
```

```
Approximate root x0 = 2.7e+00 h(x) used: 2.617867e+00
```

6. I use MATLAB to calculate the result. By following the derivations, we obtain the Jacobian matrix and implement it in the code:

$$f(X) = \begin{bmatrix} f_1(x_1, x_2) \\ f_2(x_1, x_2) \end{bmatrix} = \begin{bmatrix} \cos^2(x_1) - x_2 \\ x_1^2 + x_2^2 - x_1 - 2 \end{bmatrix} = 0$$

$$J(X) = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} \end{bmatrix} = \begin{bmatrix} -2\cos x_1 \sin x_1, & -1 \\ 2x_1 - 1, & 2x_2 \end{bmatrix}$$

```
% Define system of equations
equations = @(x) [cos(x(1))^2 - x(2); x(1)^2 + x(2)^2 - x(1) - 2];

% Define Jacobian matrix
jacobian = @(x) [-2*cos(x(1))*sin(x(1)), -1; 2*x(1) - 1, 2*x(2)];

% Define initial guess
x0 = [0.5; 0.5]; % can change

% Tolerance and maximum number of iterations
tol = 1e-8;
max_iter = 100;

% Perform Newton's method
[root, iter] = newton(equations, jacobian, x0, tol, max_iter);

% Display result
fprintf('Root of the system: [x, y] = [%d, %d]\n', root(1), root(2));
fprintf('Number of iterations: %d\n', iter);
```

```
function [root, iter] = newton(f, J, x0, tol, max_iter)
    iter = 0;
    x = x0;
    while iter < max_iter
        s = -J(x) \ f(x);    % "x = A\b" equal to solve Ax = b;
        x = x + s;
        if norm(s) < tol
            root = x;
            return;
        end
        iter = iter + 1;
    end
    error(['Newton''s method did not converge ' ...
        'within the maximum number of iterations.']);
end
```

By change initial guess we get two solutions of this system.

```
>> Q6
Root of the system: [x, y] = [1.990759e+00, 1.662412e-01]
Number of iterations: 10
>> Q6
Root of the system: [x, y] = [-9.644169e-01, 3.247816e-01]
Number of iterations: 8
```