For the source code for the coding problems, see the attached Jupyter notebook, Matlab live script. Alternatively, the entire git repository is attached as a zip archive, and is available on GitHub. The comments in the code have been omitted here for brevity. They are present in the Jupyter notebook and Matlab live script.

(1) **Problem Statement:** How many iterations would it take to find one of the roots of $f(x) := (x-1)^2 - 1$ on the interval $[1, 3]$ the bisection method with an error smaller than 10^{-5}?

The least upper bound of the absolute error for the bisection method on a given function after n iterations is given by $|x_n - c| \leq \frac{b-a}{2^n}$, where x_n is the prediction for the root at the nth iteration of the bisection method. Following this, we can solve for

$$\begin{aligned} 10^{-5} &\geq \frac{3-1}{2^n} \\ 10^{-5} \times 2^n &\geq 2 \\ 2^n &\geq 2 \times 10^5 \\ nlog(2) &\geq log(2) + 5log(10) \\ 0.30103n &\geq 5.30103 \\ n &\geq 17.60964 \\ n \in \mathbb{N} &\implies n = 18 \end{aligned}$$

However, if one were to perform the bisection method on this function on this interval, they would find that after the first iteration, the algorithm has found the exact root, meaning that technically in this instance, only one iteration was needed to get an absolute error of less than 10^{-5}.

(2) **Problem Statement:** Consider the equation $f(x) := x^2 - 2x - 3 = 0$ which has the roots $c_1 = -1$ and $c_2 = 3$. Using as interval $[a, b] = [0, 5]$, what is the approximation to c_2 using the bisection method after 3 iterations?

```
bisection_method(lambda x:  x**2 - 2*x - 3, a=0, b=5, N=3)
```

The approximation for c_2 using the bisection method after 3 iterations is 3.125.

(3) **Problem Statement:** Consider the equation $f(x) := x^2 - 2x - 8 = 0$ which has the roots $c_1 = -2$ and $c_2 = 4$. We would like to find a function so that the fixed point method $x_{n+1} = g(x_n)$ converges to the root $c_2 = 4$ for every point $x_0 \in [3, 5]$. Which $g(x)$ should we choose?

(a) $g(x) = 2 + \frac{8}{x}$ should be chosen because it converges the fastest. (b) converges to 0. Not only is this not the root we are looking for, it is not a root of $f(x)$ at all. Similarly, (d) converges to -2 immediately, which, while it is a root, is not the root we are looking for. (c) does not converge at all. That leaves only (a), which converges quickly to 4.

(4) **Problem Statement:** Write a `MATLAB` function, called `bisection_method` that takes as inputs a function, f, two numbers, a, b, an error tolerance `tol`, and a maximum number of iterations, N, and finds a root c of f in the interval $[a, b]$ using the bisection method.

```python
from typing import Callable

def bisection_method(f: Callable,
    a: float, b: float, tol: float,
     N: int = 100) -> (float, int,
    float):
     numbers = [int, float]
     if(type(a) not in numbers or
    type(b) not in numbers or type(
    tol) not in numbers):
         print("a, b, and tol must
    be numbers")
         return
     if(type(N) != int or N < 1):
         print("N must be a natural
     number.")
         return
     if(not callable(f)):
         print("f(x) must be
    callable (e.g., a function)")
         return
     if(f(a) * f(b) > 0):
         return

     err = (b - a) / 2
     for i in range(N):
         c = (a + b) / 2

         if(err < tol):
             return(c, i + 1, err)
         if(f(c) == 0):
             return(c, i + 1, err)
         if(f(a) * f(c) < 0):
             b = c
         else:
             a = c
         err = (b - a) / 2

     print("Maximum iterations
    exceeded.")
     return(c, i, err)
#
#
#
#
#
#
#
#
#
```

LISTING 1. Python

```matlab
clc; close all; clear variables;

function [c, iter, err] =
    bisection_method(f, a, b, tol,
    N)
     if ~isnumeric(a) || ~isnumeric
    (b) || ~isnumeric(tol)
         fprintf('a, b, and tol
    must be numbers');
         return;
     end
     if ~isnumeric(N) || N < 1 ||
    mod(N, 1) ~= 0
         fprintf('N must be a
    natural number.');
         return;
     end
     if ~isa(f, 'function_handle')
         fprintf('f(x) must be
    callable (e.g., a function)');
         return;
     end
     if f(a) * f(b) > 0
         return;
     end

     err = (b - a) / 2;
     for i = 1:N
         c = (a + b) / 2;

         if err < tol
             iter = i;
             return;
         end
         if f(c) == 0
             iter = i;
             return;
         end
         if f(a) * f(c) < 0
             b = c;
         else
             a = c;
         end
         err = (b - a) / 2;
     end

     fprintf('Maximum iterations
    exceeded.');
     iter = i;
     return;
end
```

LISTING 2. Matlab

(c) **Problem Statement:** Use the code to solve $f(x) = (2x^3 + 3x - 1)cos(x) - x = 0$ on $[-1, 1]$, for an accuracy of 10^{-5}. Discuss the results.

(i) The number of iterations used was 18.

(ii) The absolute error was approximately 7.6294×10^{-6}.

(iii) The expected rate of convergence for the bisection method can be expressed as $|x_n - c| \leq \frac{b-a}{2^n}$, where $n \in \mathbb{N}$. The big-O notation for this is $O(2^{-n})$.

(iv)

```python
import matplotlib.pyplot as
    plt
import numpy

f = lambda x: (2*x**3 + 3*x -
    1)*numpy.cos(x) - x

xs = numpy.linspace(-1, 1,
    400)
y = f(xs)
plt.plot(xs, y)
plt.axhline(0, color="black",
    linewidth=0.5)
plt.show()
```

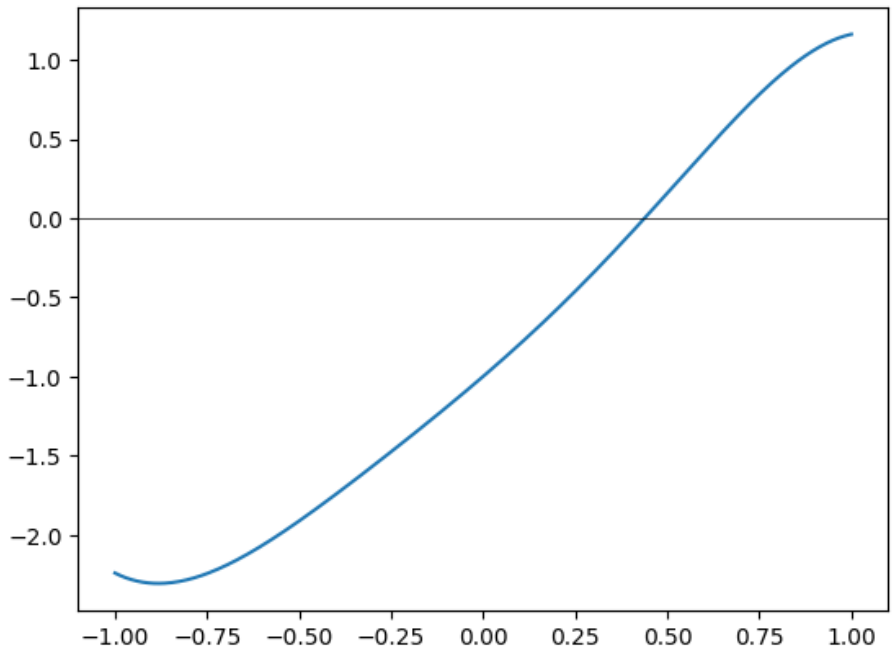
LISTING 3. Python

```matlab
clc; close all; clear
    variables;

f = @(x) (2*x.^3 + 3*x - 1).*
    cos(x) - x;

xs = linspace(-1, 1, 400);
y = f(xs);
figure;
plot(xs, y);
hold on;
yline(0, 'Color', 'black', '
    LineWidth', 0.5);
%
```

LISTING 4. Matlab



(v)

```python
from scipy.optimize import
    root_scalar
import numpy

f = lambda x: (2*x**3 + 3*x -
    1)*numpy.cos(x) - x

interval = [-1, 1]

result = root_scalar(f,
    bracket=interval)

result
```

LISTING 5. Python

```matlab
clc; close all; clear
    variables;

f = @(x) (2*x^3 + 3*x - 1)*cos
    (x) - x;

interval = [-1, 1];

root = fzero(f, interval)
%
%
%
%
```

LISTING 6. Matlab

The number computed with the Python solver is approximately 0.43857, whereas the one computed with the bisection method is approximately 0.43856. I am not sure precisely which of the available solvers in SciPy is used by default, but it is likely not the bisection method and instead a method that converges faster. That combined with the fact that there is no maximum iteration number by default means that the SciPy answer is likely closer to the answer. The fact that they vary by about 10^{-5} supports the idea that the early termination in `bisection_method` contributes to the discrepancy, since that is the least upper bound for absolute error bound that was chosen.

(5) Write a `MATLAB` function, called `fixed_point_method` that takes as inputs a function, g, an initial guess x_0, an error tolerance, `tol`, and a maximum number of iterations, N, and outputs the fixed point of g, obtained using the fixed point method, starting with x_0. It should have an error defined by $E = |x_n - x_{n-1}|$, which is the absolute difference between the last two iterations, and stop when that error is less than the tolerance, or if the number of iterations exceeds N - whichever happens first. use it to find the solution to the equation $x = e^{-x}$, with an accuracy of 10^{-10} for $x \in [-1, 1]$. State the initial guess, and how many iterations it took. Plot on the samge graph, $y = g(x)$ and $y = x$ using the 'hold on' command.

```python
def fixed_point_method(g: Callable
    , x_0: float = 0, tol: float =
    1e-5, N: int = 100) -> (float,
    int, float):
     if(not callable(g)):
         print("g(x) must be
    callable (e.g., a function).")
         return
     numbers = [int, float]
     if(type(x_0) not in numbers or
     type(tol) not in numbers):
         print("x_0 and tol must be
     numbers.")
         return
     if(type(N) != int or N < 1):
         print("N must be a natural
     number")
         return

     for i in range(N):
         print(x_0)
         c = g(x_0)
         err = abs(c - x_0)
         if(abs(c - x_0) < tol):
             return(c, i, err)
         x_0 = c

     print("Maximum iterations
    exceeded")
     return(x, i, err)
#
#
#
#
#
#
#
#
#
```

LISTING 7. Python

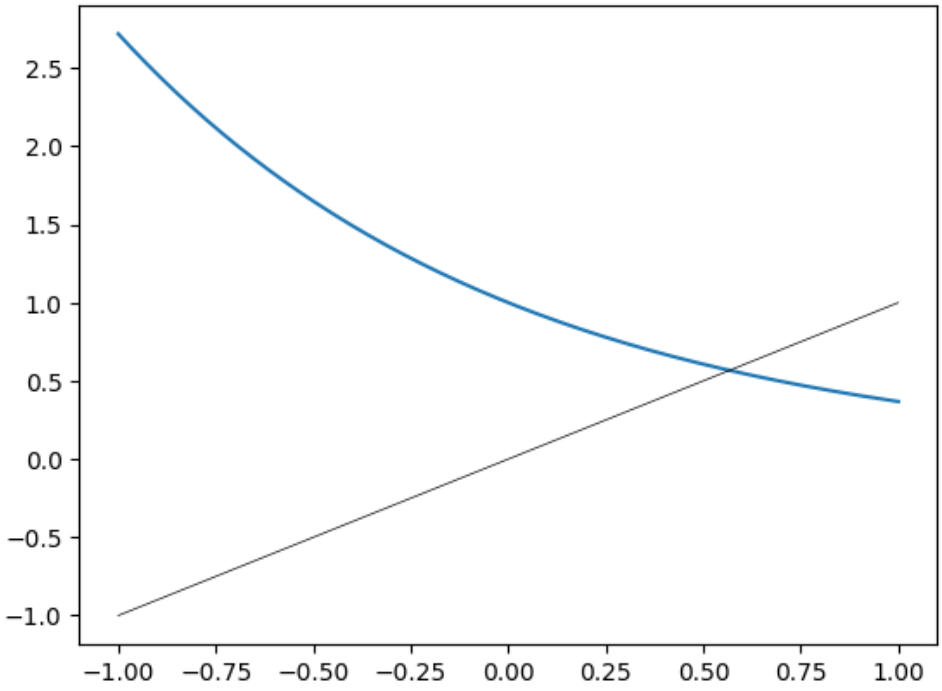
```matlab
clc; close all; clear variables;

function [c, iter, err] =
    fixed_point_method(g, x_0, tol,
     N)
     if ~isa(g, 'function_handle')
         fprintf('g(x) must be
    callable (e.g., a function).');
         return;
     end
     if ~isnumeric(x_0) || ~
    isnumeric(tol)
         fprintf('x_0 and tol must
    be numbers.');
         return;
     end
     if ~isnumeric(N) || N < 1 ||
    mod(N, 1) ~= 0
         fprintf('N must be a
    natural number.');
         return;
     end
     for i = 1:N
         fprintf(x_0);
         c = g(x_0);
         err = abs(c - x_0);
         if err < tol
             iter = i;
             return;
         end
         x_0 = c;
     end

     fprintf('Maximum iterations
    exceeded');
     iter = i;
     err = abs(c - x_0);
     return;
end
```

LISTING 8. Matlab

`fixed_point_method(lambda x: math.exp(x*-1, tol=1e-10))`
The initial guess was 0 and it took 42 iterations to converge to an absolute error of less than 10^{-10}.



(6) **Problem Statement:** Write a `MATLAB` function, called `Newtons_method` that takes as inputs a function f, its derivative, f', an initial guess x_0, an error tolerance, tol, and a maximum number of iterations, N, and outputs the root of f obtained uing Newton's method (denoted by c). It should stop when the upper bound for absolute error exceeds the error tolerance, or if the number of iterations exceeds N - whichever happens first. Use the function to find the root of the equation $arctan(x) = 1$ with an initial guess $x_0 = 2$, to an accuracy of less than `tol` $= 10^{-8}$. Did it converge? If so, in how many iterations? If not, why didn't it converge, and what happened-did it diverge, or end up in an infinite loop? Plot on the same graph the function and the eaxis $y = 0$. Test with $x_0 = -2$. What is happening?

```python
def newtons_method(f: callable, fp
    : callable, x_0: float = 0, tol
    : float = 1e-5, n: int = 100)
    -> (float, int, float):
     for i in range(n):
         try:
             f_prime = fp(x_0)
         except OverflowError:
             print("function
    diverges")
             return(float("inf"), i
    , float("inf"))
         except Exeption as e:
             print(f"Other error: {
    e}")
             return
         j = 0

         c = x_0 - f(x_0) / f_prime
         err = abs(c - x_0)

         if(abs(err) < tol):
             return(c, i + 1, err)
         if(f(c) == 0):
             return(c, i + 1, err)
         x_0 = c
     print("Maximum iterations
    exceeded")
     return(x_n, i, err)
#
#
#
#
#
#
#
#
#
#
#
#
#
#
```

LISTING 9. Python

```matlab
clc; close all; clear variables;

function [c, iter, err] =
    newtons_method(f, fp, x_0, tol,
     n)
     for i = 1:n
         try
             f_prime = fp(x_0);
         catch ME
             if strcmp(ME.
    identifier, 'MATLAB:overflow')
                 fprintf('Function
    diverges');
                 c = Inf;
                 iter = i;
                 err = Inf;
                 return;
             else
                 fprintf(['Other
    error: ', ME.message]);
                 return;
             end
         end

         c = x_0 - f(x_0) / f_prime
    ;
         err = abs(c - x_0);

         if err < tol
             iter = i;
             return;
         end
         if f(c) == 0
             iter = i;
             return;
         end
         x_0 = c;
     end
     fprintf('Maximum iterations
    exceeded');
     iter = i;
     return;
end
```

LISTING 10. Matlab

When finding the root of the equation $arctan(x) = 1$ using `Newtons_method` using $x_0 = -2$, the function runs into an overflow error. What happens is that the method diverges; the derivative at the initial point is shallow enough to cause it to choose a new point on the next iteration that is much farther away. Because with *arctan*, the derivative gets closer to 0 as x gets farther away, the distance from the starting point keeps increasing, getting further and further away from the origin until x_n is too large to be represented in the computer's memory.

Choosing $x_0 = 2$ allows the function to converge because the derivative at that point is steep enough to send the next iteration to a point closer to zero, allowing the method to converge.