Mario L. Gutierrez Abed
364009832
mlg3843@rit.edu

Problem Set 1
Numerical Analysis I

09-14-2020

Problem 1. *Find a positive root of* $x^2 - 4x\sin x + (2\sin x)^2 = 0$ *accurate to two significant figures. Use a hand calculator.*

Solution. Since I do not own a hand calculator at the moment, I instead resorted to use Mathematica as a calculator using its built-in `FindRoot` function, which internally uses numerical methods for starting at an initial value for the independent variable and locating a solution. Thus, starting at, say $x = 2$, we get

```
In[1]:= FindRoot[x^2 - 4*x*Sin[x] + (2*Sin[x])^2, {x, 2}]
Out[1]= {x -> 1.89549}
```

Hence, up to two significant digits, our root is 1.9. Here is a plot of the function to visualize the situation:

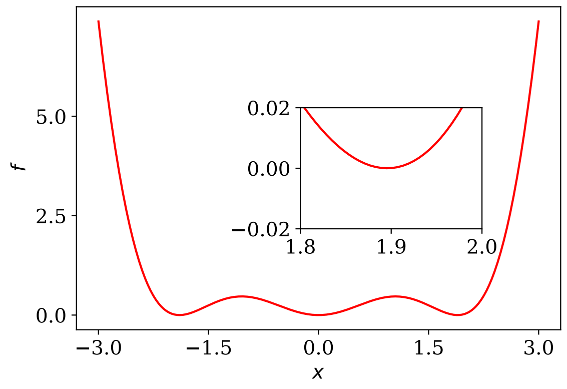

Figure 1: The function $f = x^2 - 4x\sin x + (2\sin x)^2$ plotted in the interval $[-3, 3]$. The embedded subfigure shows a zoom-in of f in the interval [1.8,2], so that we can clearly visualize the root $x \approx 1.9$. $\square$

Problem 2. *For Eqs.* (1)-(2), *use the* Intermediate Value Theorem *to find an interval of length one that contains a root of the equation. Apply two steps of the bisection method to find an approximate root within* $1/8$ *of the true root.*

$$x^5 + x = 1; \tag{1}$$
$$\sin x = 6x + 5. \tag{2}$$

Solution. Let us consider first Eq. (1). Since $f(x) = x^5 + x - 1$ is continuous on, say, the interval $[0, 1]$, it realizes every value between $f(0)$ and $f(1)$; in other words, there must be some $c \in [0, 1]$ for which $f(c) \in [f(0), f(1)]$. In fact, since $f(0) \cdot f(1) = -1 \cdot 1 = -1$, the function changes sign in this interval and therefore there must be a root $r \in [0, 1]$. Thus $[0, 1]$ is a suitable interval of length one which contains a root. As for Eq. (2), the same continuity argument can be made as for Eq. (1), and whence the Intermediate Value Theorem applies. However, this time we should use another interval to look for a root, since the values of $g(x) = \sin x - 6x - 5$ at the endpoints $\{0, 1\}$ have the same sign: $g(0) \cdot g(1) \approx -5 \cdot (-10) = 50$. Instead we shall use $[-1, 0]$, where $g(-1) \cdot g(0) \approx 0.16 \cdot (-5) = -0.8$.

Now let us iterate the bisection method for two steps, [1] so that we may find an approximate root within $1/2^{2+1} = 1/8$ of the true root r for both functions:

- $f(x) = x^5 + x - 1$, with starting endpoints $a = 0$ and $b = 1$.
  - Define the midpoint $c = (a + b)/2 = (0 + 1)/2 = 1/2$;
  - Test whether c is a root: $f(c) = f(1/2) = -15/32 \neq 0$; $\times$
  - Test whether there is a change in signs between the function evaluated at the midpoint and at one of the original endpoints, say, b: $f(b) \cdot f(c) = f(1) \cdot f(1/2) = 1 \cdot (-15/32) = -15/32$;
  - Since f changes signs in the interval $[c, b]$, the root must be here. Thus we relabel $a = c$; [End of Iteration 1]

[1] We will write a code for this on Problem 4 to get more accurate results; for now we do it by hand since that is the point of this exercise.

- Define the new midpoint $c = (a+b)/2 = (1/2+1)/2 = 3/4$;
- Test whether c is a root: $f(c) = f(3/4) = -13/1024 \neq 0$; $\times$
- Test whether there is a change in signs between the function evaluated at the newly defined midpoint and at one of the endpoints, say, b: $f(b) \cdot f(c) = f(1) \cdot f(3/4) = 1 \cdot (-13/1024) = -13/1024$;
- Since f changes signs in the interval $[c, b]$, the root must be here. Thus we relabel $a = c$; [End of Iteration 2]

· $g(x) = \sin x - 6x - 5$, with starting endpoints $a = -1$ and $b = 0$. Proceeding as we just did for f,

  - Let $c = (a+b)/2 = (-1+0)/2 = -1/2$;
  - $g(c) = g(-1/2) \approx -2.48 \neq 0$; $\times$
  - Test $g(a) \cdot g(c) = g(-1) \cdot g(-1/2) \approx 0.16 \cdot (-2.48) \approx -0.39$;
  - Thus, root is in $[a, c]$; we relabel $b = c$; [End of Iteration 1]
  - Define new midpoint $c = (a+b)/2 = (-1+(-1/2))/2 = -3/4$;
  - $g(c) = g(-3/4) \approx -1.18 \neq 0$; $\times$
  - Test $g(a) \cdot g(c) = g(-1) \cdot g(-3/4) \approx 0.16 \cdot (-1.18) \approx -0.19$;
  - Thus, root is in $[a, c]$; we relabel $b = c$; [End of Iteration 2]

Hence, from these results we obtained the approximate roots $r_f = 3/4 = 0.75$ and $r_g = -3/4 = -0.75$ for f and g, respectively. As we shall see on Problem 4 when we write our code, r_f is actually pretty close to the actual root of f, while r_g could use a bit more refinement. Alas, we have only implemented two steps of the bisection algorithm here; with more iterations, and/or with a smaller ϵ (see code for the latter parameter), we can achieve even better results. $\square$

Problem 3. *Suppose that the bisection method with starting interval $[-2, 1]$ is used to find a root of the function $f(x) = 1/x$. Does the method converge to a real number? Is it the root?*

Solution. We apply the Bisection code that we present on Problem 4 (see below) to $1/x$ on the interval $[-2, 1]$:

```
Bisect(f, -2.0, 1.0, 0.001, 19);
```

and we see that the method converges to 0. However, 0 is far from being a root! In fact $x = 0$ is an asymptote to this function, as it represents a singularity. This problem serves as a warning that we must exercise caution when applying a numerical method; we must also make sure that the results actually make sense. In this case f is continuous everywhere, except at the singularity 0 where it is not well defined; therefore the bisection method fails in this interval.

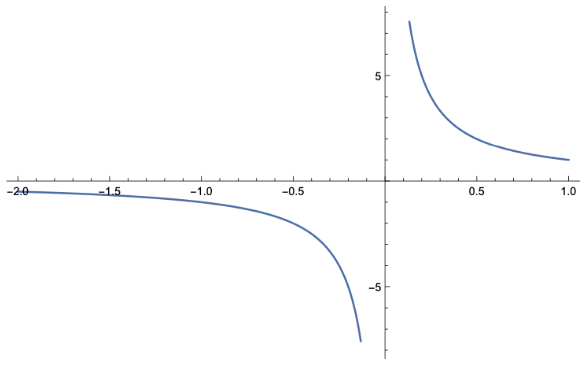

Figure 2: Plot of the function $f = 1/x$ on the interval $[-2, 1]$. Here we can see how badly the bisection method fails. $\square$

Problem 4. *Consider the equations (1)-(2) given in Problem 2. Use the bisection method to find the root to eight correct decimal places*

Solution. Here we will obtain a refinement over the results we obtained in Problem 2. Before proceeding to presenting our code, however, we need to determine how many iterations are required in order to reach a solution that is correct within eight decimal places, as the problem demands. We recall that a solution is correct within d decimal places if the error is less than 0.5×10^{-d}; we also use the fact that, for the bisection method, the error after n steps is bounded by $(b-a)/2^{n+1}$. In the problem at hand, $b-a=1$ and $d=8$, so we need to determine n, the number of necessary iterations:

$$\begin{aligned} \frac{1}{2^{n+1}} &< 0.5 \times 10^{-8} \\ -(n+1)\log 2 &< \log(0.5 \times 10^{-8}) \\ n &> -\frac{\log(0.5 \times 10^{-8})}{\log 2} - 1 \approx 26.6. \end{aligned}$$

Thus, it will take 27 iterations to reach a solution that is correct within eight decimal places. Here is the C++ code which I wrote:

```cpp
#include <iostream>
#include <fstream>
#include <cmath>

using namespace std;

double f (double x){
    return pow(x,5) + x - 1.0;
}

double g (double x){
    return sin(x) - 6.0 * x - 5.0;
}


template <typename T>
int sign (const T &val) {
    return (val > 0) - (val < 0);
}


void Bisect (double (*func)(double), double a, double b, double epsilon, int itmax){
/*
    *func = pointer reference to whatever function we want to bisect
    a,b = endpoints
    epsilon = minimum size of interval that we are willing to test
    itmax = max number of bisection iterations we are willing to test
*/
    double c {}; //initialization of variable c
    int it {0};  //iteration variable initialized to 0

    if (sign(func(a)) == sign(func(b))) {
        cout << "Signs at the endpoints are equal.
                There's no root in this interval, puny human." << endl;
    } else {
            while (b-a >= epsilon){
                 c = (a+b)/2.0; //definition of c as the midpoint
                 if (func(c) == 0.0){
                     cout << "Alas! We have found the (exact) root to be c = "
                      << c << " at func = " << func(c) << "." << endl;
                     break;
                 }
                 else if (sign(func(a)) != sign(func(c))) {
                        b = c;
                 } else {
                        a = c;
                 }
                if (it == itmax) {
                    cout << "We have reached the max number of bisection iterations that
                    we are willing to test. No exact root found; our best approximation is
                    c = " << c << " at func = " << func(c) << "." << endl;
                    break;
                }
                it += 1;
            }
            if (b-a < epsilon) {
                cout << "We have reached the minimum size of interval that
                we are willing to test. No exact root found; our best approximation is
                c = " << c << " at func = " << func(c) << "." << endl;
            }
        }
}

int main(int argc, const char * argv[]) {

cout << setprecision(8);    //set output precision to 8 significant digits

Bisect(f, 0.0, 1.0, 1e-10, 26);
Bisect(g, -1.0, 0.0, 1e-10, 26);

    return 0;
}
```

To clarify, the function call

```
Bisect(f, 0.0, 1.0, 1e-10, 26);
```

means that we are evaluating the function f, in the interval $[0, 1]$, with a tolerance $\epsilon = 1^{-10}$ for the minimum interval size that we are willing to let our code test; [2] lastly, the last parameter means that we are only allowing the code to run for up to twenty-seven iterations. [3] In the end, the code will stop running if either we find an exact root $c = 0$, or if we reach the minimum interval size ($< \epsilon$), or if the maximum number of iterations is reached. In the latter two cases, of course, we end up with a (good!) approximation to the root (for "nice" functions anyway ... may the example from Problem 3 serve as a warning).

The code runs for exactly twenty-seven iterations, and then yields the following output:

```
We have reached the max number of bisection iterations that we are willing to test. No exact
     root found; our best approximation is c = 0.75487766 at func = -5.5039903e-09.
We have reached the max number of bisection iterations that we are willing to test. No exact
     root found; our best approximation is c = -0.97089892 at func = -2.552891e-08.
Program ended with exit code: 0
```

Hence we have obtained the (approximate) roots $r_f = 0.75487766$ and $r_g = -0.97089892$, for f and g, respectively, which are correct to eight decimal places. Just for the fun of it, let us see how our results stack up when compared to the output you get from using Mathematica's built-in function FindRoot, which we used in Problem 1. Starting f at $x = 1$ and g at $x = 0$, we get

```
In[1]:=
f[x_] := x^5 + x - 1;
g[x_] := Sin[x] - 6*x - 5;

SetPrecision[FindRoot[f[x], {x, 1}], 8]
SetPrecision[FindRoot[g[x], {x, 0}], 8]

Out[3]= {x -> 0.75487767}
Out[4]= {x -> -0.97089892}
```

These values are indeed in agreement with the results obtained from using our very own code! We close out this exercise by plotting the functions and taking a closer look at the roots we found:

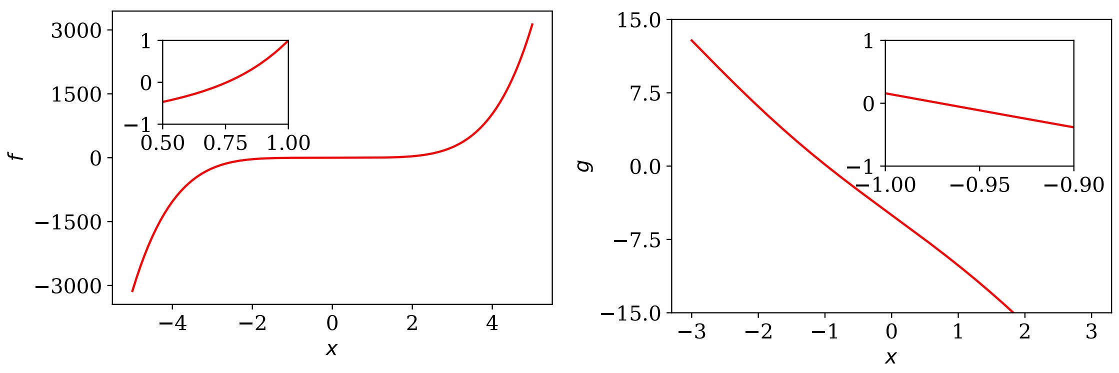

Figure 3: Plots of both $f = x^5 + x - 1$ and $g = \sin x - 6x - 5$. Embedded subfigures show a zoom-in of the functions in the vicinity of the roots r_f and r_g that we found in this exercise. □

[2] I picked a very small ϵ in this case to make sure that the code doesn't break until all 27 iterations have passed.
[3] Unlike Matlab, in C++ the count always starts at 0.

Problem 5. *Consider the equations (1)-(2) given in Problem 2. Apply fixed-point iteration (FPI) to find the solution of each equation to eight correct decimal places.*

Solution. Starting with Eq. (1), we may be tempted to rewrite as $x = 1 - x^5$ and then define $g_1(x) \equiv 1 - x^5$. However, by cheating a bit and looking at the root we obtained in the previous problem, we see that

$$g_1'(0.75487767) \approx -1.6,$$

whose absolute value is greater than 1. Thus we know that by using this g_1 the FPI method will fail badly. We could instead rewrite Eq. (1) as $x = \sqrt[5]{1-x}$ and set $g_1(x) \equiv \sqrt[5]{1-x}$; this will yield a more satisfactory

$$g_1'(0.75487767) \approx -0.6,$$

whose absolute value is less than 1, and thus we would get convergence. However, we can do even better . . . using a similar trick as the one we discussed in the lectures, we add $4x^5$ to both sides of Eq. (1), and then rearranging terms and isolating x we get

$$x = \frac{1+4x^5}{1+5x^4}. \tag{3}$$

Now, defining g_1 as the RHS of (3), and evaluating the derivative g_1' at the root, we get

$$g_1'(0.75487767) \approx 0.$$

This is as good as it gets! By reducing the slope of the iteration function g_1 we get a more rapidly convergent FPI implementation. Let us see it in practice in the following straightforward C++ code I wrote:

```cpp
#include <iostream>
#include <fstream>
#include <cmath>

using namespace std;

double g (double x){
    return (1.0 + (4.0 * pow(x,5)) )/( 1.0 + (5.0 * pow(x,4)) ) ;
}

void FPI (double (*func)(double), double x0, int itmax){
    double x {x0};    //initialization of variable x to our initial guess x0
    int it {0};
    for (int i {0}; i <= itmax; i++){
        x = func(x);
        it += 1;
        cout << "Iteration # " << it << "." << endl;
        if (abs(func(x) - x) <= 5.0 * 1e-8) {
            break;
        }
    }
    cout << "The solution (that is, the fixed point) is x = " << x << "." << endl;
}

int main(int argc, const char * argv[]) {

    cout << setprecision(8);
    FPI(g, 1.0, 10);

    return 0;
}
```

Let us briefly explain a couple of things in this code. In the function call

```cpp
    FPI(g, 1.0, 10);
```

the first argument is our function $g_1 = (1+4x^5)/(1+5x^4)$, as defined at the beginning of the snippet shown. The second argument is our initial guess x_0, which we set to 1 in this case, and lastly the third argument is the maximum number of iterations that we are willing to let the code run. The number 10 is arbitrary here, of course, but since we want to find the solution correct to eight decimal places the key part of the code is, in fact,

```cpp
    if (abs(func(x) - x) <= 5.0 * 1e-8) {
            break;
        }
```

Here we are telling the machine to stop the loop if at some iteration we arrive at the condition $|g_1(x) - x| \leq 5 \times 10^{-8}$. In fact, since this particular g_1 we chose has a derivative with an almost non-existent slope, we expect it to converge after just a few iterations. Indeed, look at the output of our code:

```
Iteration # 1.
Iteration # 2.
Iteration # 3.
Iteration # 4.
Iteration # 5.
The solution (that is, the fixed point) is x = 0.75487767.
Program ended with exit code: 0
```

This shows that after just five iterations we found our fixed-point solution, and ...voilà!...it is the same root we had previously found using the bisection method. For comparison, running the algorithm for $g_1(x) = \sqrt[5]{1-x}$ (which, as we saw earlier, is also convergent) took 37 iterations(!) to arrive at the solution with the same 10^{-8} accuracy. Of course, in order to find the best choice for g_1 we cheated and used our knowledge of the root from the previous problem. In practice, we will not know the root a priori (since that is what we are trying to find in the first place!), so we can only make our best smart guess and then check to see how large the absolute value of the derivative is. Note, however, that although we did use the root from the previous problem as a crutch to find the best choice for g_1, we did not use that root at all in our code; hence our algorithm is valid regardless.

Moving on to Eq. (2), we rewrite it as $x = (\sin x - 5)/6$, and then define $g_2(x) \equiv (\sin x - 5)/6$. Again, we cheat and use the previously obtained root to check whether our FPI implementation will converge, or if we are just wasting our time with this choice for g_2 and we need something better...In fact, this choice of g_2 is fantastic:

$$g_2'(-0.97089892) \approx 0.09 \ll 1. \quad \surd$$

Applying the same code as before to g_2,

```
    FPI(g, 0.0, 10);
```

we get convergence after just eight iterations:

```
Iteration # 0.
Iteration # 1.
Iteration # 2.
Iteration # 3.
Iteration # 4.
Iteration # 5.
Iteration # 6.
Iteration # 7.
Iteration # 8.
The solution (that is, the fixed point) is x = -0.97089892.
Program ended with exit code: 0
```

If I had started at $x_0 = -1$ I would have gotten even faster convergence (six iterations), but I feel like we've been cheating too often with our knowledge of the roots fron the previous problems! ☺ □

Problem 6. *Derive three different $g(x)$ for finding roots to six correct decimal places of $f(x) = 2x^3 - 6x - 1$ by FPI (that is, convert it to the form $x = g(x)$). Run FPI for each $g(x)$ and report results, convergence or divergence. The equation $f(x) = 0$ has three roots. Derive more $g(x)$ if necessary until all roots are found by FPI. For each convergent run, determine the value of S from the errors e_{n+1}/e_n, and compare with S from calculus (that is, compare it with $S = |g'(r)|$).*

Solution. We set $f(x) = 0$ and rewrite it in the following three ways

$$x = \frac{1}{3}x^3 - \frac{1}{6} \equiv g_1(x); \tag{4a}$$

$$x = \sqrt[3]{3x + \frac{1}{2}} \equiv g_2(x); \tag{4b}$$

$$x = \frac{1}{2(x^2 - 3)} \equiv g_3(x). \tag{4c}$$

We now report results for all three functions, using the same FPI code we have written before, except that now we also ask the code to give us the values x_i at each step, since we want to determine the value of S from the errors at the end.

● For g_1, starting with initial guess $x_0 = 0.3$,

```
FPI(g, 0.3, 100);
```

we get excellent results:

```
Iteration # 0. x = 0.3.
Iteration # 1. x = -0.157667.
Iteration # 2. x = -0.167973.
Iteration # 3. x = -0.168246.
The solution (that is, the fixed point) is x = -0.168254.
Program ended with exit code: 0
```

It only took three iterations to find our fixed point solution to six correct decimal places! That means that S must be very small; let us see now both variants of this value:

$$S = \lim_{i\to\infty} \frac{e_{i+1}}{e_i} \approx \frac{e_3}{e_2} = \frac{|x_3 - r|}{|x_2 - r|} = \frac{|-0.168246 - (-0.168254)|}{|-0.167973 - (-0.168254)|} = 0.0284698; \tag{5a}$$

$$S = |g_1'(r)| = |g_1'(-0.168254)| = 0.0283094. \tag{5b}$$

● We now attempt g_2, starting with initial guess $x_0 = 1.3$,

```
FPI(g, 1.3, 100);
```

Here we also get convergence, although not as fast as with g_1 (but still, quite good!):

```
Iteration # 0. x = 1.3.
Iteration # 1. x = 1.638643.
Iteration # 2. x = 1.756134.
Iteration # 3. x = 1.793433.
Iteration # 4. x = 1.804955.
Iteration # 5. x = 1.808485.
Iteration # 6. x = 1.809564.
Iteration # 7. x = 1.809893.
Iteration # 8. x = 1.809994.
Iteration # 9. x = 1.810024.
Iteration # 10. x = 1.810034.
Iteration # 11. x = 1.810037.
The solution (that is, the fixed point) is x = 1.810038.
Program ended with exit code: 0
```

This time it took eleven iterations to find our fixed point solution to six correct decimal places. Thus we expect S to have a larger value than that of g_1:

$$S \approx \frac{|x_{11} - r|}{|x_{10} - r|} = \frac{|1.810037 - 1.810038|}{|1.810034 - 1.810038|} = 0.25; \tag{6a}$$

$$S = |g_2'(1.810038)| = 0.305228. \tag{6b}$$

We notice some larger disparity now between the two variants of S, although this can commonly happen due to the truncation of $\lim_{i\to\infty}$ that must happen in a finite grid …The important thing here is that both variants show $S < 1$, which ensures convergence.

● On to g_3, starting with initial guess $x_0 = -1.3$,

```
FPI(g, -1.3, 100);
```

This one also yield fantastic results, but we have a problem…the root we find with g_3 is the same one we found earlier with g_1:

```
Iteration # 0. x = -1.3.
Iteration # 1. x = -0.381679.
Iteration # 2. x = -0.175173.
Iteration # 3. x = -0.168389.
Iteration # 4. x = -0.168257.
The solution (that is, the fixed point) is x = -0.168254.
Program ended with exit code: 0
```

It merely took four iterations to find our fixed point solution to six correct decimal places. Thus S value must be very small; indeed,

$$S \approx \frac{|x_4 - r|}{|x_3 - r|} = \frac{|-0.168257 - (-0.168254)|}{|-0.168389 - (-0.168254)|} = 0.0222222; \tag{7a}$$

$$S = |g_3'(-0.168254)| = 0.0190528. \tag{7b}$$

We are still missing one root; let us attempt another fixed-point function, g_4, and see if it yields the missing root. We rewrite $f(x) = 0$ as $2x^3 - 6x = 1$ and add $3x^3$ to both sides to get

$$x = \frac{3x^3 + 1}{5x^2 - 6} \equiv g_4(x). \tag{8}$$

Then, with initial guess $x_0 = -2$,

```
FPI(g, -2.0, 100);
```

we get an excellent result that also happens to find our missing root!:

```
Iteration # 0. x = -2.
Iteration # 1. x = -1.642857.
Iteration # 2. x = -1.641398.
Iteration # 3. x = -1.641923.
Iteration # 4. x = -1.641733.
Iteration # 5. x = -1.641802.
Iteration # 6. x = -1.641777.
Iteration # 7. x = -1.641786.
Iteration # 8. x = -1.641783.
The solution (that is, the fixed point) is x = -1.641784.
Program ended with exit code: 0
```

After eight iterations we found our missing fixed point solution to six correct decimal places. Let us check on the S values:

$$S \approx \frac{|x_8 - r|}{|x_7 - r|} = \frac{|-1.641783 - (-1.641784)|}{|-1.641786 - (-1.641784)|} = 0.5; \tag{9a}$$

$$S = |g_2'(-1.641784)| = 0.360485. \tag{9b}$$

One peculiar aspect of g_4 is that, unlike our previous choices, it actually yields all three roots(!) depending on where you start the algorithm. For instance, had we started with initial guess $x_0 = 0$,

```
FPI(g, 0.0, 100);
```

we get convergence to the root $r = -0.168254$ after just two iterations!:

```
Iteration # 0. x = 0.
Iteration # 1. x = -0.166667.
Iteration # 2. x = -0.168246.
The solution (that is, the fixed point) is x = -0.168254.
Program ended with exit code: 0
```

Had we instead started at $x_0 = 2$,

```
FPI(g, 2.0, 100);
```

we get convergence to the root $r = 1.810038$ after ten iterations:

```
Iteration # 0. x = 2.
Iteration # 1. x = 1.785714.
Iteration # 2. x = 1.818478.
Iteration # 3. x = 1.807462.
Iteration # 4. x = 1.810859.
Iteration # 5. x = 1.80978.
Iteration # 6. x = 1.81012.
Iteration # 7. x = 1.810012.
Iteration # 8. x = 1.810046.
Iteration # 9. x = 1.810035.
Iteration # 10. x = 1.810039.
The solution (that is, the fixed point) is x = 1.810038.
Program ended with exit code: 0
```

Anyhow, we have found all three roots of $f(x) = 2x^3 - 6x - 1$ by using our very own FPI code. We conclude by showing a graph of f, showcasing all three roots we just found.

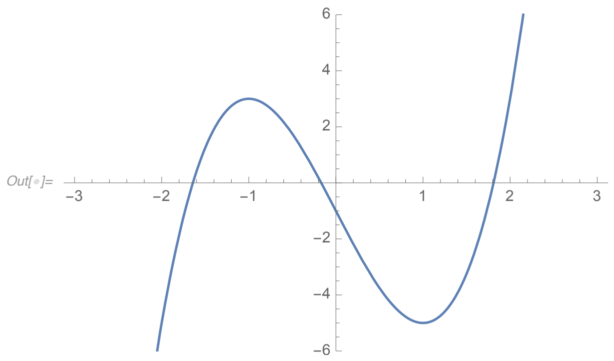

Figure 4: The function $f = 2x^3 - 6x - 1$ and its three roots, $\{-1.641784, -0.168254, 1.810038\}$. □

Problem 7. *Consider the equations (1)-(2) given in Problem 2. Use Newton's Method to find the root to eight correct decimal places.*

Solution. Time to write another code! For Newton's Method we need both the function and its derivative. Starting with Eq. (1),

$$f(x) = x^5 + x - 1; \tag{10a}$$
$$f'(x) = 5x^4 + 1, \tag{10b}$$

while for Eq. (2), we have

$$g(x) = \sin x - 6x - 5; \tag{11a}$$
$$g'(x) = \cos x - 6. \tag{11b}$$

Here is our C++ code that implements Newton's Method, and applies it to the functions from (1) and (2):

```cpp
#include <iostream>
#include <fstream>
#include <cmath>

using namespace std;

double f (double x){
    return pow(x,5) + x - 1.0;
}

double fder (double x){
    return 5.0 * pow(x,4) + 1.0;
}

double g (double x){
    return sin(x) - (6.0 * x) - 5.0;
}

double gder (double x){
    return cos(x) - 6.0;
}


void Newton (double (*func)(double), double (*funcder)(double), double x0,
             double epsilon = 0.5 * 1e-8, int itmax = 100){
    double x {x0};    //initialization of variable x to our initial guess x0
    double newx {};   //initialization of variable newx

    for (int i {0}; i <= itmax; i++){
        cout << "Iteration # " << i << "." << " x = " << x << "." << endl;
        newx = x - ( func(x) )/( funcder(x) );
        if (abs(newx - x) <= epsilon) {
            cout << "Newton Method has converged within given tolerance.
                     The root found is x = " << newx << "." << endl;
            break;
        }
        if (i == itmax) {
            cout << "Newton Method has failed to converge after the maximum
                    allowed number of iterations. " << endl;
        }
        x = newx;
    }
}

int main(int argc, const char * argv[]) {

    cout << setprecision(8);
    Newton(f, fder, 0);
    Newton(g, gder, 0);

    return 0;
}
```

The results are excellent; we find the roots quite quickly; for f:

```
Iteration # 0. x = 0.
Iteration # 1. x = 1.
Iteration # 2. x = 0.83333333.
Iteration # 3. x = 0.76438212.
Iteration # 4. x = 0.75502487.
Iteration # 5. x = 0.7548777.
Iteration # 6. x = 0.75487767.
Newton Method has converged within given tolerance. The root found is x = 0.75487767.
```

and for g:

```
Iteration # 0. x = 0.
Iteration # 1. x = -1.
Iteration # 2. x = -0.97096377.
Iteration # 3. x = -0.97089892.
Newton Method has converged within given tolerance. The root found is x = -0.97089892.
```

□

Problem 8. *A* 10cm*-high cone contains* $60\,\text{cm}^3$ *of ice cream, including a hemispherical scoop on the top. Find the radius of the scoop to four correct decimal places.*

Solution. Assuming that the cone is completely filled with ice cream (this is a necessary assumption), then finding the radius of the scoop is pretty straightforward, since

$$\begin{aligned} V_{\text{total}} &= V_{\text{cone}} + V_{\text{scoop}} \\ 60 &= \frac{10\pi r^2}{3} + \frac{4\pi r^2}{3} \cdot \frac{1}{2} \\ r &= \sqrt{\frac{15}{\pi}} \approx 2.2. \end{aligned}$$

The problem is asking us to solve the problem on a computer/calculator though, so let's just plug in Mathematica:

```
In[6]:=
SetPrecision[Solve[(60 == (\[Pi]*r^2*10)/3 + (2*\[Pi]*r^2)/3) && r >= 0, r], 5]
Out[6]= {{r -> 2.1851}}
```

□

Problem 9. *Consider the function* $f(x) = e^{\sin^3 x} + x^6 - 2x^4 - x^3 - 1$ *on the interval* $[-2, 2]$*. Plot the function on the interval, and find all three roots to six correct decimal places. Determine which roots converge quadratically, and find the multiplicity of the roots that converge linearly.*

Solution. Here is the graph of the function:

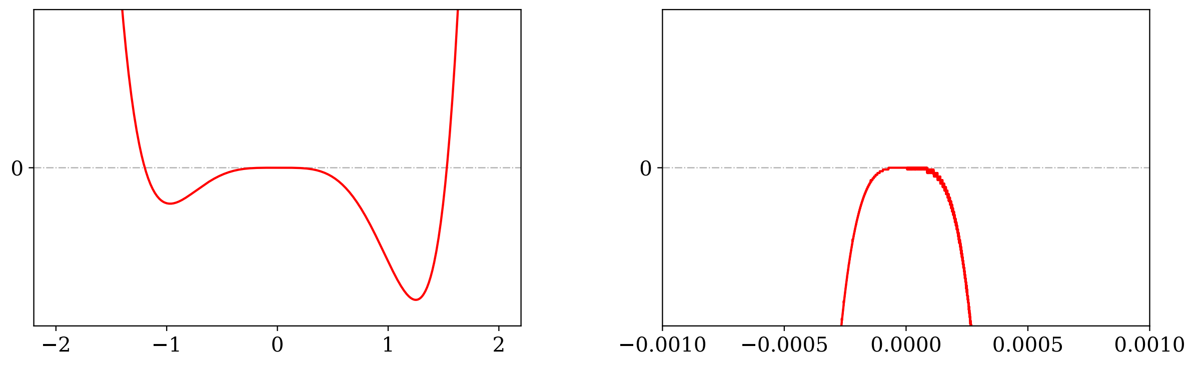

Figure 5: Graph of $f(x) = e^{\sin^3 x} + x^6 - 2x^4 - x^3 - 1$ on the interval $[-2, 2]$ (left). We can see that it has three roots, one of which has nontrivial multiplicity. In fact, from the figure on the left one may be mislead to think that the function vanishes in a whole interval surrounding $x = 0$, but as the zoom-in on the right figure shows, that is not the case.

We can tell from just looking at the plot that Newton's Method will fail badly if we start with initial guess $x_0 = 0$, since the tangent (and hence the derivative) there vanishes and, consequently, Newton's Method attempts to divide by zero. The algorithm is divergent in this case, albeit we can apply it in the vicinity and get convergence. Before tackling the more complicated root $x = 0$, let us look at the other two, for which we easily get convergence:

```
double f (double x){
    return exp( pow(sin(x), 3) ) + pow(x,6) - ( 2.0 * pow(x,4) ) - pow(x,3) - 1.0;
}

double fder (double x){
    return ( cos(x) * pow(sin(x), 2) * 3.0 * exp( pow(sin(x), 3) ) ) + ( 6.0 * pow(x,5) ) -
    ( 8.0 * pow(x,3) ) - ( 3.0 * pow(x,2) );
}

int main(int argc, const char * argv[]) {

    cout << setprecision(7);
    Newton(f, fder, -1.0, 0.5 * 1e-6);

    return 0;
}
```

which has output

```
Iteration # 0. x = -1.
Iteration # 1. x = -2.221536.
Iteration # 2. x = -1.896474.
Iteration # 3. x = -1.637268.
Iteration # 4. x = -1.438844.
Iteration # 5. x = -1.300688.
Iteration # 6. x = -1.224206.
Iteration # 7. x = -1.199918.
Iteration # 8. x = -1.197643.
Iteration # 9. x = -1.197624.
Newton Method has converged within given tolerance. The root found is x = -1.197624.
Program ended with exit code: 0
```

Thus, with starting guess $x_0 = -1$, Newton's Method has found the root $r = -1.197624$ after just nine iterations. If we instead choose initial guess $x_0 = 2$, we get another root after just six iterations:

```
Iteration # 0. x = 2.
Iteration # 1. x = 1.779275.
Iteration # 2. x = 1.629974.
Iteration # 3. x = 1.552496.
Iteration # 4. x = 1.531542.
Iteration # 5. x = 1.53014.
Iteration # 6. x = 1.530134.
Newton Method has converged within given tolerance. The root found is x = 1.530134.
Program ended with exit code: 0
```

These two are cases of *simple roots*, since $f'(r) \neq 0$ in either case. The function is also twice continuously differentiable, with $f''(r)$ having values

```
In[1]:=
f[x_] := E^(Sin[x])^3 + x^6 - 2 x^4 - x^3 - 1;
fprime[x_] := D[f[x], x];
D[fprime[x], x] /. x -> -1.197624
D[fprime[x], x] /. x -> 1.530134

Out[3]= 35.6295
Out[4]= 91.0323
```

In such situations, we get very fast (that is, *quadratic*) convergence, as the following theorem from Sauer's text states:

Theorem 1.11 (Sauer)

Let f be twice continuously differentiable and $f(r) = 0$. If $f'(r) \neq 0$, then Newton's Method is locally and quadratically convergent to r. The error e_i at step i satisfies

$$\lim_{i\to\infty} \frac{e_{i+1}}{e_i^2} = M < \infty, \tag{12}$$

where $e_i = |x_i - r|$ is the error at the i^{th} step, as before, and

$$M = \left| \frac{f''(r)}{2f'(r)} \right| . \tag{13}$$

Let us evaluate this M value (both the exact value and the truncated error ratio) for both single roots:

- For $r = -1.197624$,

$$M \approx \frac{e_8}{e_7^2} = \frac{|-1.197643 - (-1.197624)|}{(|-1.199918 - (-1.197624)|)^2} \approx 3.6 \tag{14a}$$

$$M = \left| \frac{f''(-1.197624)}{2f'(-1.197624)} \right| = \left| \frac{35.6295}{2 \cdot (-4.92059)} \right| \approx 3.6. \tag{14b}$$

- For $r = 1.530134$,

$$M \approx \frac{e_5}{e_4^2} = \frac{|1.530140 - 1.530134|}{(|1.531542 - 1.530134|)^2} \approx 3.03 \tag{15a}$$

$$M = \left| \frac{f''(1.530134)}{2f'(1.530134)} \right| = \left| \frac{91.0323}{2 \cdot (14.9728)} \right| \approx 3.04. \tag{15b}$$

Now, for the remaining root we apply our code with initial guess $x_0 = 0.5$,

```
Newton(f, fder, 0.5, 0.5 * 1e-6);
```

The output now (showing only the last couple of iterations) is

```
Iteration # 28. x = 0.000155763.
Iteration # 29. x = 0.0001227148.
Iteration # 30. x = 8.516709e-05.
Newton Method has converged within given tolerance. The root found is x = 8.516709e-05.
Program ended with exit code: 0
```

We see that the method is converging to the root $x = 0$; due to our set precision, the value $8.516709 \times 10^{-5} \approx 0.000085$ is as close as we can get.[4] However, note that this time it took 30 iterations to converge to a solution within the given tolerance, while for the previous two roots the method was much faster. This is an example of a *multiple root*; i.e., a root for which $f'(r) = 0$. In fact, we can see that $x = 0$ is a root of f of multiplicity 4, since $0 = f(0) = f'(0) = f''(0) = f^{(3)}(0)$, but $f^{(4)}(0) = -48 \neq 0$; we show this in the following Mathematica snippet:

```
In[1]:=
f[x_] := E^(Sin[x])^3 + x^6 - 2 x^4 - x^3 - 1;
fprime[x_] := D[f[x], x];
fpprime[x_] := D[fprime[x], x];
fppprime[x_] := D[fpprime[x], x];
fpppprime[x_] := D[fppprime[x], x];

fprime[x] /. x -> 0
fpprime[x] /. x -> 0
fppprime[x] /. x -> 0
fpppprime[x] /. x -> 0

Out[6]= 0
Out[7]= 0
Out[8]= 0
Out[9]= -48
```

In this case, the following theorem from Sauer's applies:

[4] Reaching ***exactly*** zero is impossible for a computer with finite precision!

Theorem 1.12 (Sauer)

Assume that f is C^{m+1} on an interval (a, b) and has a root r of multiplicity m in such interval. Then Newton's Method is locally (linear) convergent to r, and the errors satisfy

$$\lim_{i\to\infty} \frac{e_{i+1}}{e_i} = S, \tag{16}$$

where

$$S = \frac{m-1}{m}. \tag{17}$$

Let us evaluate the S value (both the exact value and the truncated error ratio) for our root $x = 0$:

$$S \approx \frac{e_{29}}{e_{28}} = \frac{|0.0001227148 - 0|}{|0.000155763 - 0|} \approx 0.79 \tag{18a}$$

$$S = \frac{4-1}{4} = \frac{3}{4} = 0.75. \tag{18b}$$

How could we have improved our results when dealing with the complications brought about by the $x = 0$ root? Very easily actually, thanks to the *Modified Newton Method*:

Theorem 1.13 (Sauer)

If f is C^{m+1} on an interval (a, b) and has a root r of multiplicity $m > 1$ in such interval. Then the Modified Newton Method

$$x_{i+1} = x_i - \frac{mf(x_i)}{f'(x_i)} \tag{19}$$

converges locally and quadratically(!) to r.

Putting eq. (19) to good use in our code, with $m = 4$, we get an incredible boost in performance! Check out the insane difference!:

```
Iteration # 0. x = 0.5.
Iteration # 1. x = -0.03166195.
Iteration # 2. x = 7.533888e-05.
Newton Method has converged within given tolerance. The root found is x = 7.533888e-05.
Program ended with exit code: 0
```

We see that this modified version of Newton's Method converges to the root $x = 0$ after just two iterations! ☐

Problem 10. *Use the Secant Method to find the (single) solution of* $x^3 = 2x + 2$.

Solution. The code for the Secant Method is just a slight variation of the Newton Method code we presented above, the two noticeable differences being that now we need not one, but two initial points, and that this time we do not require to know the derivative of the function. Here is a snippet of the code:

```
#include <iostream>
#include <fstream>
#include <cmath>

using namespace std;

double f (double x){
    return pow(x,3) - (2.0 * x) - 2.0;
}

void Secant (double (*func)(double), double xo, double x0,
             double epsilon = 0.5 * 1e-8, int itmax = 100){
    double x {x0};    //initialization of variable x to our initial guess x0
    double prevx  {x0}; //initialization of variable prevx to our initial guess x0
    double newx {};

    for (int i {0}; i <= itmax; i++){
        cout << "Iteration # " << i << "." << " x = " << x << "." << endl;
        newx = x - (  func(x) * (x - prevx) )/( func(x) - func(prevx) );
        if ( abs(newx - x) <= epsilon ) {
            cout << "Secant Method has converged within given tolerance.
                    The root found is x = " << newx << "." << endl;
            break;
        }
        if (i == itmax) {
            cout << "Secant Method has failed to converge after the maximum
                    allowed number of iterations. " << endl;
        }
        prevx = x;
        x = newx;
    }
}


int main(int argc, const char * argv[]) {

    cout << setprecision(7);
    Secant(f, 0.0, 1.0);

    return 0;
}
```

The output shows convergence to the root $x = 1.769292$ after twelve iterations:

```
Iteration # 0. x = 0.
Iteration # 1. x = -2.
Iteration # 2. x = 1.
Iteration # 3. x = 4.
Iteration # 4. x = 1.157895.
Iteration # 5. x = 1.296255.
Iteration # 6. x = 2.253636.
Iteration # 7. x = 1.610618.
Iteration # 8. x = 1.722752.
Iteration # 9. x = 1.775239.
Iteration # 10. x = 1.769089.
Iteration # 11. x = 1.769291.
Iteration # 12. x = 1.769292.
Secant Method has converged within given tolerance. The root found is x = 1.769292.
Program ended with exit code: 0
```

□

Problem 11. *Consider the iteration formula*

$$x_{n+1} = x_n - \frac{f(x_n)}{g(x_n)}, \tag{20a}$$

where

$$g(x) = \frac{f(x + f(x)) - f(x)}{f(x)}. \tag{20b}$$

Show that this is quadratically convergent under suitable hypotheses.

Proof. We assume that r is a root solution to this iteration formula and that $f'(r) \neq 0$, since otherwise r would be a multiple root and we would most certainly not have quadratic convergence. Now, with that out of the way, let us start by getting rid of that awkward $f(x + f(x))$ term in $g(x)$. By Taylor's Theorem, there exists ξ_n between x_n and $x_n + f(x_n)$ such that

$$f(x_n + f(x_n)) = f(x_n) + f(x_n)f'(x_n) + \frac{1}{2}f^2(x_n)f''(\xi_n).$$

Plugging this into (20b), we get

$$g(x_n) = f'(x_n) + \frac{1}{2}f''(\xi_n)f(x_n).$$

Thus we rewrite Eq. (20a) as

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n) + \frac{1}{2}f''(\xi_n)f(x_n)}. \tag{21}$$

Now we must try to find a relation between the error at the $n+1$ step, namely $e_{n+1} = x_{n+1} - r$ and the error at the n step, $e_n = x_n - r$.[5] To that end, we substract r from both sides of (21) so that we get

$$e_{n+1} = e_n - \frac{f(x_n)}{f'(x_n) + \frac{1}{2}f''(\xi_n)f(x_n)}. \tag{22}$$

Now, since we want to show quadratic convergence, we want to find at some point the ratio e_{n+1}/e_n^2. Let us now attempt that; using the fact that $f(r) = 0$, we Taylor-expand again, this time at the root:

$$0 = f(r) = f(x_n - e_n) = f(x_n) - f'(x_n)e_n + \frac{1}{2}f''(\Xi_n)e_n^2$$

for some choice of Ξ_n between r and x_n. Hence, we have found that

$$f(x_n) = f'(x_n)e_n - \frac{1}{2}f''(\Xi_n)e_n^2. \tag{23}$$

Plugging this into (22) and expanding,

$$\begin{aligned} e_{n+1} &= e_n - \frac{f'(x_n)e_n - \frac{1}{2}f''(\Xi_n)e_n^2}{f'(x_n) + \frac{1}{2}f''(\xi_n)f(x_n)} \\ &= \frac{e_n\left[f'(x_n) + \frac{1}{2}f''(\xi_n)f(x_n)\right] - f'(x_n)e_n + \frac{1}{2}f''(\Xi_n)e_n^2}{f'(x_n) + \frac{1}{2}f''(\xi_n)f(x_n)} \\ &= \frac{\frac{1}{2}f''(\xi_n)f(x_n)e_n + \frac{1}{2}f''(\Xi_n)e_n^2}{f'(x_n) + \frac{1}{2}f''(\xi_n)f(x_n)} \\ &= \frac{f''(\xi_n)\left[f'(x_n)e_n - \frac{1}{2}f''(\Xi_n)e_n^2\right]e_n + f''(\Xi_n)e_n^2}{2f'(x_n) + f''(\xi_n)f(x_n)} \\ &= e_n^2\frac{f''(\xi_n)\left[f'(x_n) - \frac{1}{2}f''(\Xi_n)e_n\right] + f''(\Xi_n)}{2f'(x_n) + f''(\xi_n)f(x_n)} \\ &= e_n^2\frac{f''(\xi_n)f(x_n) + f''(\Xi_n)e_n}{g(x_n)e_n}. \end{aligned} \tag{24}$$

Letting

$$\Psi_n \equiv \frac{f''(\xi_n)f(x_n) + f''(\Xi_n)e_n}{g(x_n)e_n},$$

and M being some finite number, we see that the iteration formula (20a) is quadratically convergent if we make the suitable hypotheses that

$$\lim_{n\to\infty}\frac{e_{n+1}}{e_n^2} = \lim_{n\to\infty}\Psi_n = M < \infty. \qquad \square$$

Problem 12. *Which of the following sequences converge quadratically?*

(a) $\frac{1}{n^2}$

(b) $\frac{1}{2^{2^n}}$

(c) $\frac{1}{\sqrt{n}}$

(d) $\frac{1}{e^n}$

(e) $\frac{1}{n^n}$

[5] We are dropping, WLOG, the absolute value sign in order to make the calculations more readable.

Solution. A sequence $\{x_n\}$ is quadratically convergent if

$$\lim_{n\to\infty} \frac{x_{n+1}}{x_n^2} < \infty. \tag{25}$$

Let us check this condition on each of the above sequences:

- For (a),

$$\lim_{n\to\infty} \frac{\frac{1}{(n+1)^2}}{\frac{1}{n^4}} = \lim_{n\to\infty} \frac{n^4}{n^2+2n+1} = \lim_{n\to\infty} \frac{n^2}{1+\frac{2}{n}+\frac{1}{n^2}} = \lim_{n\to\infty} n^2 = \infty \qquad \times$$

- For (b),

$$\lim_{n\to\infty} \frac{\frac{1}{2^{2^{n+1}}}}{\frac{1}{(2^{2^n})^2}} = \lim_{n\to\infty} \frac{2^{2^n\cdot 2}}{2^{2^n\cdot 2}} = 1 < \infty \qquad \surd$$

- For (c),

$$\lim_{n\to\infty} \frac{\frac{1}{\sqrt{n+1}}}{\frac{1}{n}} = \lim_{n\to\infty} \frac{n}{\sqrt{n+1}} \times \frac{\frac{1}{\sqrt{n}}}{\frac{1}{\sqrt{n}}} = \lim_{n\to\infty} \frac{\sqrt{n}}{\sqrt{1+\frac{1}{n}}} = \lim_{n\to\infty} \sqrt{n} = \infty \qquad \times$$

- For (d),

$$\lim_{n\to\infty} \frac{\frac{1}{e^{n+1}}}{\frac{1}{(e^n)^2}} = \lim_{n\to\infty} \frac{e^{2n}}{e^{n+1}} = \lim_{n\to\infty} e^{n-1} = \infty \qquad \times$$

- Lastly, for (e), we need to work harder:

$$\begin{aligned}
\lim_{n\to\infty} \frac{\frac{1}{(n+1)^{n+1}}}{\frac{1}{(n^2)^{n^2}}} &= \lim_{n\to\infty} \frac{n^{2n^2}}{(n+1)^n(n+1)} \times \frac{\frac{1}{n^n}}{\frac{1}{n^n}} \\
&= \lim_{n\to\infty} \frac{(n^{2n^2-n})^n}{[(1+\frac{1}{n})^n(n+1)]^n} \\
&= \lim_{n\to\infty} \frac{n^{2n^3-n^2}}{\left[(1+\frac{1}{n})^n\right]^n (n+1)^n} \times \frac{\frac{1}{n^n}}{\frac{1}{n^n}} \\
&= \lim_{n\to\infty} \frac{n^{2n^3-n^2-n}}{\left[(1+\frac{1}{n})^n\right]^n (1+\frac{1}{n})^n} \\
&= \frac{1}{e} \lim_{n\to\infty} \frac{\log_n[n^{2n^3-n^2-n}]}{\log_n[e^n]} \\
&= \frac{1}{e} \lim_{n\to\infty} \frac{2n^3-n^2-n}{\log_n e^n} \\
&= \frac{1}{e} \lim_{n\to\infty} \frac{n^3\left(2-\frac{1}{n}-\frac{1}{n^2}\right)}{\frac{\ln e^n}{\ln n}} \\
&= \frac{1}{e} \lim_{n\to\infty} n^2 \ln n \left(2-\frac{1}{n}-\frac{1}{n^2}\right) = \infty \qquad \times
\end{aligned}$$

Thus, we have determined that the only quadratically-convergent sequence from above is the one from part (b). □

Problem 13. *Show that Newton's iteration will diverge for $f_1(x) = x^2+1$ and $f_2(x) = 7x^4+3x^2+\pi$, no matter what (real) starting point is selected.*

Proof. Since both f_1 and f_2 have even powers of x, and we are assuming that $x \in \mathbb{R}$, a quick look at both f_1 and f_2 should convince you that they are nonzero and positive, no matter what x we decide to evaluate the functions on. Therefore, neither function will ever equal zero, and hence there is no (real) root to be found. That is the logic argument, but now we will show this using Newton's Method,

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}. \tag{26}$$

Applying (26) to f_1,

$$x_{n+1} = x_n - \frac{x_n^2+1}{2x_n} = \frac{x_n^2-1}{2x_n}. \tag{27}$$

Let us assume that, as we continue to iterate over and over, i.e., as $n \to \infty$, the sequence (x_n) converges to some limit ℓ. Then from (27), we would get

$$\ell = \frac{\ell^2-1}{2\ell} \implies \ell^2+1=0 \implies \ell = \pm i \notin \mathbb{R}. \tag{28}$$

Thus we have shown that convergence of f_1 to a real number is not possible.

We approach f_2 similarly:

$$x_{n+1} = x_n - \frac{7x_n^4+3x_n^2+\pi}{28x_n^3+6x_n} = \frac{21x_n^4+3x_n^2-\pi}{28x_n^3+6x_n}. \tag{29}$$

Assuming, as above, that the sequence (x_n) converges to some limit ℓ, we get

$$\ell = \frac{21\ell^4+3\ell^2-\pi}{28\ell^3+6\ell} \implies 7\ell^4+3\ell^2+\pi=0 \implies \ell = \pm\sqrt{\frac{-\pi}{7\ell^2+3}}. \tag{30}$$

But since $7\ell^2+3$ must necessarily be nonnegative if $\ell \in \mathbb{R}$, the whole expression inside the radical is negative, which in turn implies that $\ell \in \mathbb{C}$. Hence, there is no real root for f_2 either. □

Problem 14. *Show that the function* $f(x) = 2 + x - \arctan(x)$ *has the property* $|f'(x)| < 1$*. Show that* f *does not have a fixed point. Explain why this does not contradict the* Contraction Mapping Theorem.

Solution. Taking the derivative of f, we get

$$f'(x) = 1 - \frac{1}{1+x^2} = \frac{x^2}{1+x^2}. \tag{31}$$

Since $x \in \mathbb{R}$, the denominator of this expression will always be larger (by 1) than the numerator; therefore $f'(x)$ (and hence $|f'(x)|$) must be less than 1. Assume now that f has a fixed point $\bar{x}$, so that

$$f(\bar{x}) = f\left(\lim_{n\to\infty} x_n\right) = \lim_{n\to\infty} f(x_n) = \lim_{n\to\infty} x_{n+1} = \bar{x}.$$

Then,

$$\begin{aligned}\lim_{n\to\infty} f(x_n) &= \lim_{n\to\infty}[2 + x_n - \arctan(x_n)]\\ \lim_{n\to\infty} x_{n+1} &= \lim_{n\to\infty}[2 + x_n - \arctan(x_n)]\\ \bar{x} &= 2 + \bar{x} - \arctan(\bar{x})\\ \arctan(\bar{x}) &= -2.\end{aligned}$$

But this is nonsensical, since $-\pi/2 < \arctan < \pi/2$, and $-2 < -\pi/2$, so it is outside of arctan's range. Thus, we conclude that f cannot possibly have a fixed point.

We show now that this result just obtained does not contradict the *Contraction Mapping Theorem*, which states that if f is a contraction mapping, then it has a unique fixed point. Let us first recall that a ***contraction***, or a ***contraction mapping*** is a map $f\colon (X,d) \to (X,d)$, where (X,d) is a metric space with metric d, that satisfies

$$d(f(x), d(y)) \le \lambda d(x,y); \qquad 0 < \lambda < 1.$$

In our case, we are dealing with the real line and the standard Euclidean metric, so we may rewrite this as a map $f\colon [a,b] \to [a,b]$, where $[a,b] \in \mathbb{R}$, that satisfies

$$|f(x) - f(y)| < \lambda |x - y|. \tag{32}$$

By the *Intermediate Value Theorem*, there is some $\xi \in [a,b]$ that satisfies

$$\frac{f(x)-f(y)}{x-y} = f'(\xi).$$

We rewrite this as

$$f(x)-f(y) = f'(\xi)(x-y),$$

and then take absolute values on both sides:

$$\begin{aligned} |f(x)-f(y)| &= |f'(\xi)(x-y)| \\ &= |f'(\xi)||(x-y)| \\ &< |x-y|. \end{aligned} \tag{33}$$

The last inequality comes from the fact that $|f'(\eta)| < 1$ for any $\eta \in [a,b]$, as we discussed at the beginning of this exercise. Note that the inequality (33) is a more relaxed condition than the one given by the expression (32) for a contraction mapping. If f were a contraction, then the inequality (32) would have to hold for any $\lambda \in (0,1)$. What we have found is that f satisfies thre more relaxed condition (33) instead. Hence, we may conclude that f is not a contraction mapping, and thus it is not to be held accountable for the Contraction Mapping Theorem. □

Problem 15. *Let $f\colon \mathbb{R} \to \mathbb{R}$ be a given function.*

(a) *Write down Newton's method for approximating the square root of a number $c > 0$.*

(b) *Find a simple recursion relation for the error $e_n = x_n - \sqrt{c}$.*

(c) *Prove using the recursion from part (b) that*

  i. *if $x_0 > \sqrt{c}$, the sequence x_n will monotonically decrease to $\sqrt{c}$;*

  ii. *the convergence will be quadratic as the limit is approached.*

Solution to (a). We define $f(x) = x^2 - c$, and write down Newton's Method for this function:

$$x_{n+1} = x_n - \frac{x_n^2 - c}{2x_n} = \frac{x_n^2 + c}{2x_n}. \qquad \square \tag{34}$$

Solution to (b). From Eq. (34), we have

$$e_{n+1} = x_{n+1} - \sqrt{c} = \frac{x_n^2 + c}{2x_n} - \sqrt{c}.$$

We now write this expression in terms of e_n:

$$\begin{aligned} e_{n+1} &= \frac{x_n + \frac{c}{x_n}}{2} - \sqrt{c} \\ 2e_{n+1} &= \underbrace{x_n - \sqrt{c}}_{=e_n} + \frac{c}{x_n} - \sqrt{c} \\ 2e_{n+1} &= e_n + \frac{\sqrt{c}\,\overbrace{(\sqrt{c} - x_n)}^{=-e_n}}{x_n} \\ 2e_{n+1} &= e_n - \frac{e_n\sqrt{c}}{x_n} = \frac{e_n(x_n - \sqrt{c})}{x_n} = \frac{e_n^2}{e_n + \sqrt{c}}. \end{aligned}$$

Thus we have found the error recursion relation

$$\boxed{e_{n+1} = \frac{e_n^2}{2(e_n + \sqrt{c})}} \qquad \square \tag{35}$$

Solution to (c). Showing that the sequence (x_n) is monotonically-decreasing to $\sqrt{c}$ entails proving the following two statements

$$x_n - x_{n+1} \geq 0; \tag{36a}$$
$$\lim_{n\to\infty} x_n = \sqrt{c}. \tag{36b}$$

Proving (36a) using the recursion relation (35) took me down an induction path that I really don't want to type ... it's a bit too messy for my taste. Instead, I shall use a much simpler observation. From Eq. (34) we get

$$x_n - x_{n+1} = \frac{x_n^2 - c}{2x_n}. \tag{37}$$

Expanding the numerator of the RHS and, again, using (34), we have

$$\begin{aligned}
x_n^2 - c &= \left(\frac{x_{n-1} + \frac{c}{x_{n-1}}}{2}\right)^2 - c \\
&= \frac{1}{4}\left(x_{n-1}^2 + 2c + \frac{c^2}{x_{n-1}^2}\right) - c \\
&= \frac{1}{4}\left(x_{n-1}^2 - 2c + \frac{c^2}{x_{n-1}^2}\right) \\
&= \frac{1}{4}\left(x_{n-1} - \frac{c}{x_{n-1}}\right)^2 \\
&\geq 0.
\end{aligned} \tag{38}$$

Thus, $x_n \geq +\sqrt{c} > 0$ (the last inequality from the assumption that $c > 0$). Hence, the RHS of (37) is nonnegative and we have established Eq. (36a); i.e., (x_n) is monotonically decreasing. Now, if the sequence approaches a limit, say ℓ, then taking the limit as $n \to \infty$ on Eq. (34),

$$\ell = \frac{\ell^2 + c}{2\ell} \implies \ell^2 - c = 0 \implies \ell = \pm\sqrt{c}. \tag{39}$$

In particular, combining the assumption that $x_0 > \sqrt{c}$ with the fact that the sequence is decreasing monotonically, the first of these two limits that the sequence will run into is $+\sqrt{c}$, and it will converge there.

Showing that such convergence is quadratic is straightforward. Note from the recursion relation (35) that

$$\begin{aligned}
\frac{e_{n+1}}{e_n^2} &= \frac{1}{2x_n} \\
\lim_{n\to\infty} \frac{e_{n+1}}{e_n^2} &= \lim_{n\to\infty} \frac{1}{2x_n} = \frac{1}{2\sqrt{c}} < \infty.
\end{aligned}$$

□

Problem 16. *Let $f\colon \mathbb{R} \to \mathbb{R}$ be a given function.*

(a) *Consider the following variation of Newton's method:*

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_0)}. \tag{40}$$

Find C and s such that

$$e_{n+1} = Ce_n^s. \tag{41}$$

(b) Halley's Method *for solving the equation $f(x) = 0$ uses the iteration formula*

$$x_{n+1} = x_n - \frac{f_n f_n'}{(f_n')^2 - \frac{f_n f_n''}{2}}. \tag{42}$$

Show that this formula results when Newton's iteration is applied to the function $f/\sqrt{f'}$.

(c) *Show that the secant method can be written in the form*

$$x_{n+1} = \frac{x_{n-1}f_n - x_n f_{n-1}}{f_n - f_{n-1}}. \tag{43}$$

Solution to (a). By substracting both sides of Eq. (40) from the root r we obtain

$$e_{n+1} = e_n - \frac{f(x_n)}{f'(x_0)}. \tag{44}$$

On the other hand,

$$0 = f(r) = f(x_n - e_n) = f(x_n) - e_n f'(\xi_n)$$

for some real ξ_n. (The last equality comes from Taylor-expanding at the root to first order.) Thus we have

$$f(x_n) = e_n f'(\xi_n),$$

which implies

$$e_{n+1} = e_n - \frac{f(x_n)}{f'(x_0)} = e_n \left(1 - \frac{f'(\xi_n)}{f'(x_0)}\right).$$

This yields

$$s = 1 \qquad \text{and} \qquad C = 1 - \frac{f'(\xi_n)}{f'(x_0)}.$$

□

Solution to (b). We have the iteration formula

$$x_{n+1} = x_n - \frac{\frac{f_n}{\sqrt{f_n'}}}{\left(\frac{f_n}{\sqrt{f_n'}}\right)'}. \tag{45}$$

Expanding the last term,

$$\begin{aligned}
\frac{\frac{f_n}{\sqrt{f_n'}}}{\left(\frac{f_n}{\sqrt{f_n'}}\right)'} &= \frac{\frac{f_n}{\sqrt{f_n'}}}{\frac{f_n'\sqrt{f_n'} - \frac{f_n f_n''}{2\sqrt{f_n'}}}{f_n'}} \\
&= \frac{f_n}{\sqrt{f_n'}} \cdot \frac{\frac{f_n'}{2(f_n')^2 - f_n f_n''}}{2\sqrt{f_n'}} \\
&= \frac{f_n f_n'}{(f_n')^2 - \frac{f_n f_n''}{2}}.
\end{aligned}$$

Plugging this back into (45) we arrive at the formula given by Halley's Method, Eq. (42). □

Solution to (c).

$$\begin{aligned}
x_{n+1} &= x_n - \frac{f_n(x_n - x_{n-1})}{f_n - f_{n-1}} \\
&= \frac{f_n x_n - f_{n-1}x_n - f_n x_n + f_n x_{n-1}}{f_n - f_{n-1}} \\
&= \frac{x_{n-1}f_n - x_n f_{n-1}}{f_n - f_{n-1}}.
\end{aligned}$$

□