1. (a) Use the bisection method to find the roots of

\[ f(x) = \sinh(x) - \frac{(1 + x^3)}{2} \]

with tolerance \( \epsilon = 10^{-15} \). Make certain that you find all of the roots of the given function. Record the final endpoints of the bracketing interval, the achieved tolerance, the number of iterations, and the computational time for each root. (You may use the prepared matlab script `bisect.m`, if you wish.)

(b) Repeat part (a) using instead the Newton-Fourier method discussed in the course lectures. You will need to write your own matlab script (e.g. by modifying `newton.m`). For each root use the same initial interval \([a, b]\) as you did for the bisection method in part (a). Note that the function may not satisfy the criteria for application of the Newton-Fourier method, for each of the roots. In that case, you will need to transform the function as discussed in class.

(c) Compare the results of the bisection and Newton-Fourier methods (final endpoints of the bracketing interval, achieved tolerance, number of iterations, and computational time). Which method is preferable and why?

(a) We use the following code to plot the graph of \( f \) over the real line:

```matlab
f=@(x) sinh(x)-(1+x.^3)/2;
figure; fplot(@(x) f(x),[0*x,-3 5])
```

![Graph of f(x) over the real line](image)

We find 4 roots, in the bracketing intervals \([-3, -2], [0, 1], [1, 2], [4, 5]\) respectively. Since \( \frac{\sinh(x)-(1+x^3)}{2} \rightarrow \infty \) as \( x \rightarrow \infty \), there are no roots for larger \( x \). We use the following code to implement the bisection root-finding method:

```matlab
tol=1e-15;
a=-3; b=-2;
bisect pause
```
To implement Newton-Fourier method we need to transform $f$. One can calculate and find
\[
    f'(x) = \cosh(x) - \frac{3x^2}{2},
\]
\[
    f''(x) = \sinh(x) - 3x.
\]
Near the first root $f' < 0, f'' > 0$: we need to set $f_{new}(x) = f(-x);$  
Near the second root $f' > 0, f'' < 0$: we set $f_{new}(x) = -f(-x);$  
Near the third root $f' < 0, f'' < 0$: we need to set $f_{new}(x) = -f(x);$  
Near the fourth root $f' > 0, f'' > 0$: we need to set $f_{new}(x) = f(x).$

Or more easily we can find the above result by looking at the graph of $f$. The slope decides the sign of the first derivative and the convexity decides the sign of the second derivative. We modify the `newton.m` to get `newfour.m`:

```matlab
 tic
 itmax=500;
 if sign(f(x))==sign(f(y))
    'failure to bracket root'
    return
 end
 k=0
 [x y y-x]
 while abs(x-y)>tol*max(abs(y),1.0)
    if k+1>itmax
       break
    end
    xold=x;
    yold=y;
    k=k+1
    DF=Df(y);
    x=x-f(x)/DF;
    y=y-f(y)/DF;
    [x y y-x]
```
We then implement this Newton-Fourier method to find the roots:

```matlab
fold=f;
fold=@(x) sinh(x)-(1+x.^3)/2;
Dfold=@(x) cosh(x)-3*x.^2/2;
figure; fplot(@(x) [fold(x), 0*x], [-3,-2])
pause
f=@(x) -sinh(x)-(1-x.^3)/2;
Df=@(x) -cosh(x)+3*x.^2/2;
x=-2; y=3;
newtfour
x=-x
pause
figure; fplot(@(x) [fold(x), 0*x], [0,1])
pause
f=@(x) sinh(x)+(1-x.^3)/2;
Df=@(x) cosh(x)-3*x.^2/2;
x=-1; y=0;
newtfour
pause
figure; fplot(@(x) [fold(x), 0*x], [1 2])
pause
f=@(x) -sinh(x)+(1+x.^3)/2;
Df=@(x) -cosh(x)+3*x.^2/2;
x=1; y=2;
newtfour
pause
figure; fplot(@(x) [fold(x), 0*x], [4 5])
pause
f=fold;
Df=Dfold;
x=4; y=5;
newtfour
```

Note we need to convert the roots for $f_{new}$ back to the roots for $f$:

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>tol</th>
<th>#iter</th>
<th>time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>-2.055090542866180</td>
<td>-2.055090542866180</td>
<td>0</td>
<td>5</td>
<td>0.001305</td>
</tr>
<tr>
<td>0.557220276222911</td>
<td>0.557220276222911</td>
<td>0</td>
<td>6</td>
<td>0.001149</td>
</tr>
<tr>
<td>1.438741118188025</td>
<td>1.438741118188025</td>
<td>0</td>
<td>7</td>
<td>0.001276</td>
</tr>
<tr>
<td>4.567345052205096</td>
<td>4.567345052205096</td>
<td>0</td>
<td>7</td>
<td>0.000857</td>
</tr>
</tbody>
</table>

(c) By the two tables in (a) and (b), both methods achieve the requested level of tolerance. The number of iterations required by bisection in these examples is roughly seven times larger than for Newton (49 versus 7) and the wall clock time is also roughly seven times larger. The Newton-Fourier method is thus superior both in computational time, and number of iterations. The cost is that one has to know the signs of the first two derivatives of the function beforehand and also compute new three function values per iteration. ■
2. (a) Use Newton’s method to find the positive root of smallest magnitude of
\[ f(x) = \frac{x^2 - 3}{x^2 + 1}, \]
again with tolerance \( \epsilon = 10^{-15} \) and \( x_0 = 2 \). Record the final approximate root, the achieved tolerance, the number of iterations, and the computational time.

(b) Repeat part (a) using the secant method. Use the same initial guess \( x_0 = 2 \) as you did for Newton’s method in (a) and make one Newton iteration to generate \( x_1 \).

(c) Repeat part (a) using the method of inverse quadratic interpolation (IQI). Use the same initial \( x_0, x_1 \) as you did for the secant method in (b), and make one secant iteration to generate \( x_2 \).

(d) Compare the results of the three methods (final endpoints of the bracketing interval, achieved tolerance, number of iterations, and computational time). Which method is preferable and why?

\[ f(x) = \frac{x^2 - 3}{x^2 + 1}, \]
\[ f'(x) = \frac{8x}{(x^2 + 1)^2}. \]

We first plot \( f \) to bracket the root:

(a) We use `newton.m` and the following code to find the root. We will average over 100 times to find a more accurate estimate of time consumed using the method.

```matlab
1 f=@(x) (x.^2-3)/(x.^2+1);
2 figure; fplot(@(x) [f(x), 0*x], [0 3])
```

![Plot of f(x) to bracket the root](image)

(a) We use `newton.m` and the following code to find the root. We will average over 100 times to find a more accurate estimate of time consumed using the method.

```matlab
1 Df=@(x) 8*x./(x.^2+1).^2;
2 tol=1e-15;
3 tic
4 for ii=1:100
5 x=2;
6 % turn off tic-toc in newton!
```
We use secant.m and the following code to find the root.

```matlab
tic
for ii=1:100
    a=2;
    clear b
    % turn off tic-toc in secant!
    secant
    end
tot=toc;
k=k
[b abs(b-a)]
time=tot/100
```

We use iquadi.m and the following code to find the root.

```matlab
tic
for ii=1:100
    a=2;
    clear b
    clear c
    % turn off tic-toc in iquadi!
    iquadi
    end
tot=toc;
k=k
[c abs(c-b)]
time=tot/100
```

<table>
<thead>
<tr>
<th>method</th>
<th>root</th>
<th>tol</th>
<th>#iter</th>
<th>time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Newton</td>
<td>1.732050807568877</td>
<td>0</td>
<td>6</td>
<td>0.0001228</td>
</tr>
<tr>
<td>Secant</td>
<td>1.732050807568877</td>
<td>0</td>
<td>7</td>
<td>0.0002836</td>
</tr>
<tr>
<td>IQI</td>
<td>1.732050807568877</td>
<td>0</td>
<td>6</td>
<td>0.0004675</td>
</tr>
</tbody>
</table>

Each method converges in comparable numbers of iterations to the same tolerance. The inverse quadratic interpolation method requires about four times the amount of time as Newton’s method, while the secant method requires only about twice as much time as Newton. Both secant method and IQI make one function evaluation in each iteration, compared with Newton’s two function evaluations, but Newton has quadratic rate of convergence and this greater convergence rate wins in the present example. Newton’s method is usually superior when the derivative of the function is easy to evaluate.

3. Derive the error formula for the secant method:

\[
x_n - x_{n+1} = -(x_n - x_{n-1})(x_n - x_{n-1}) \frac{f[x_{n-1}, x_n, x_n]}{f[x_{n-1}, x_n, x_{n-1}]}.\]

\[
x_n - x_{n+1} = x_n - x_n + f(x_n) \frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})}
\]
Plugging these into the formula for the Newton Method iterate, we see that
\[
\frac{f(x_n)}{f'[x_{n-1}, x_n]} = \frac{(x_n - x_n)x_{n-1}}{f[x_{n-1}, x_n]} + f(x_n)
\]

Then we have
\[
\frac{f(x_n)}{f'[x_{n-1}, x_n]} = \frac{(x_n - x_n)x_{n-1}}{f[x_{n-1}, x_n]} + f(x_n) - f(x_n)
\]

4. Consider Newton’s method for finding the positive square root of \( a > 0 \). Derive the following results, assuming \( x_0 > 0, x_0 \neq \sqrt{a} \).

(a) \( x_{n+1} = \frac{1}{2} \left( x_n + \frac{a}{x_n} \right) \)

(b) \( x^2 - a = \left( \frac{x^2 - a}{2x_n} \right)^2 \) for \( n \geq 0 \), and thus \( x_n > \sqrt{a} \) for all \( n > 0 \).

(c) The iterates \( \{x_n\} \) are a strictly decreasing sequence for \( n \geq 1 \). Hint: Consider the sign of \( x_{n+1} - x_n \).

(d) \( e_{n+1} = e_n^2 / (2x_n) \), with \( e_n = x_n - \sqrt{a} \) and \( \text{Rel}(x_{n+1}) = \frac{\sqrt{2}}{2x_n} |\text{Rel}(x_n)|^2 \) with \( \text{Rel}(x_n) \) the relative error in \( x_n \).

(e) Derive an inequality that relates \( \text{Rel}(x_{n+1}) \) and \( \text{Rel}(x_n) \) for any integer \( n \). Use this result to bound \( \text{Rel}(x_n) \) if \( x_0 \geq \sqrt{a} \) and \( |\text{Rel}(x_0)| \leq 0.1 \).

(a) Finding the positive square root of \( a \) is equivalent to finding the positive root of the function \( f(x) = x^2 - a \). This function has derivative \( f'(x) = 2x \).

Plugging these into the formula for the Newton Method iterate, we see that
\[
x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} = \frac{x^2_n - a}{2x_n}
\]
\[
= \frac{x_n}{2} + \frac{a}{2x_n}
\]

(b) From above, we have that
\[
x^2_{n+1} = \left( \frac{x_n}{2} + \frac{a}{2x_n} \right)^2 = \frac{1}{4} \left( x^2_n + 2a + \frac{a^2}{x^2_n} \right).
\]

Then we have
\[
x^2_{n+1} - a = \left( \frac{x^2_n - 2a + \frac{a^2}{x^2_n}}{2x_n} \right) = \left( \frac{x_n}{2} - \frac{a}{2x_n} \right)^2 = \left( \frac{x^2_n - a}{2x_n} \right)^2 > 0,
\]
since the square of any nonzero quantity is positive. Therefore, \( x_n > \sqrt{a} \) \( \forall n \geq 1 \).

(c) We investigate the difference between iterates:

\[
x_{n+1} - x_n = \frac{1}{2}(x_n + a) - x_n = \frac{a - x_n^2}{2x_n}.
\]

By part (b), since \( \sqrt{a} < x_n \), the numerator of the quantity is negative. The denominator is positive because \( x_n > \sqrt{a} > 0 \). Therefore, the difference is negative and the iterates are monotonic decreasing.

(d) From part (a),

\[
x_{n+1} - \sqrt{a} = \frac{1}{2}(x_n - 2\sqrt{a} + \frac{a}{x_n}) = \frac{1}{2x_n}(x_n - \sqrt{a})^2.
\]

Define error \( e_n = x_n - \sqrt{a} \). Then \( e_{n+1} = \frac{1}{2x_n}e_n^2 \).

We also have that

\[
\text{Rel}(x_{n+1}) = \frac{e_{n+1}}{\sqrt{a}} = \frac{e_n^2}{2x_n\sqrt{a}} = \frac{\sqrt{a}}{2x_n}[\text{Rel}(x_n)]^2.
\]

(e) For \( x_n > \sqrt{a} \),

\[
\text{Rel}(x_{n+1}) < \frac{1}{2}[\text{Rel}(x_n)] \\
2\text{Rel}(x_{n+1}) < [2\text{Rel}(x_n)]^2 \\
2\text{Rel}(x_n) < [2\text{Rel}(x_0)]^{2^n} \\
\text{Rel}(x_n) < \frac{1}{2}[2\text{Rel}(x_0)]^{2^n}.
\]

With this result, we can bound \( \text{Rel}(x_4) \) in the situation presented in the problem:

\[
\text{Rel}(x_4) \leq \frac{1}{2}[2\text{Rel}(x_0)]^{16} \leq \frac{1}{2}(0.2)^{16} = 3.28 \times 10^{-12}.
\]

6. Newton’s method for finding a root \( x_* \) of \( f(x) = 0 \) sometimes requires the initial guess \( x_0 \) to be quite close to \( x_* \) in order to obtain convergence. Verify that this is the case for the root \( x_* = \frac{3\pi}{2} \) of

\[
f(x) = \sin(2x) + \sin(99x) - 1
\]

Based on the convergence proof, make a rough estimate how small \( |x_0 - x_*| \) must be for iterates to converge to \( x_* \). Check this estimate numerically by slowly increasing the distance of \( x_0 \) from \( x_* \) until the Newton iteration fails to converge.

Denote the interval in which we search by \( [\frac{3\pi}{2} - \epsilon, \frac{3\pi}{2} + \epsilon] \) for some small \( \epsilon > 0 \). Let

\[
M = \max_{|x - 3\pi/2| \leq \epsilon} |f''(x)|,
\]

We know that to guarantee convergence, \( |x_* - x_0| < \frac{1}{M} \), and further, \( f' \neq 0 \) in this interval. We have
\[ f'(x) = 2 \cos(2x) + 99 \cos(99x), \]
\[ f''(x) = -4 \sin(2x) - 99^2 \sin(99x). \]

Numerically we find the nearest zero of \( f' \) around \( \frac{3\pi}{2} \) to be \( 4.712184905708748 \), giving \( \delta = \left| 4.712184905708748 - \frac{3\pi}{2} \right| = 2.04074676 \times 10^{-4}. \)

We also evaluate \( f'(\frac{3\pi}{2}) = -2, f''(\frac{3\pi}{2}) = -9801 \). So for \( \epsilon \) small enough, \( M = \frac{9801}{4} \) and \( 1/M = 4.9801 \approx 4.0812162 \times 10^{-4}. \)

As a result, we require that
\[ \epsilon < \min\{\delta, \frac{1}{M}\} = \delta \approx 2.04074676 \times 10^{-4} \]

The following code can be used to check when the convergence fails:

```matlab
f=@(x) sin(2*x)+sin(99*x)-1;
Df=@(x) 2*cos(2*x)+99*cos(99*x);
DDf=@(x)-4*sin(2*x)-99^2*sin(99*x);
x=3*pi/2;
Mstar=abs(DDf(x)/Df(x)/2);
delstar=1/Mstar;
figure; hold on
plot(@(x) f(x), [3*pi/2-1/M,3*pi/2+1/M])
plot(3*pi/2,0,'r*')
y=fzero(@(x) Df(x), [3*pi/2-0.01,3*pi/2+0.01]);
plot(y,0,'b*')
Delta=abs(y-3*pi/2);
for ii=1:200
    x=3*pi/2-ii/200/M;
    newton
    check=abs(x-3*pi/2);
    if check>tol
        fprintf('failure for %0.0f \n', ii)
        fprintf('failed at distance %0.8f \n', ii/200/M)
        return
    return
end
end
```

If we run this, we see that convergence fails for the first time when \( |x_0 - x_*| = 0.00020406 \). This agrees quite well with our \textit{a priori} estimate.