1. a. We use information about the Taylor expansions for the exponential and sine functions to give:

$$\begin{aligned} f(x) &= 2e^{-2x} - 2\cos(2x) + 4\sin(x) \\ f(x) &= 2\left(1 - 2x + \frac{4x^2}{2!} - \frac{8x^3}{3!}\right) - 2\left(1 - \frac{4x^2}{2!}\right) + 4\left(x - \frac{x^3}{3!}\right) + R_3(x) \\ f(x) &= 8x^2 - \frac{10}{3}x^3 + R_3(x), \qquad \text{where} \quad P_3(x) = 8x^2 - \frac{10}{3}x^3. \end{aligned}$$

where

$$R_3(x) = \frac{f^{(4)}(\xi(x))x^4}{4!} = \frac{(32e^{-2\xi(x)} - 32\cos(2\xi(x)) + 4\sin(\xi(x)))x^4}{4!}, \qquad \xi(x) \in [0, x).$$

b. For $x \in [0, 1]$, we have

$$|R_3(x)| \le \frac{(32 + 32 + 4)}{4!} = \frac{17}{6} \approx 2.8333.$$

(The optimal bound is 0.87556, since $f^{(4)}(1) = 21.0133$ is largest.) We have $f(1) = 4.468848$ and $P_3(1) = \frac{14}{3} = 4.666667$, so the absolute error and relative errors are

$$\begin{aligned} |P_3(1) - f(1)| &= 0.19782 \\ \frac{|P_3(1) - f(1)|}{|f(1)|} &= 0.044266, \end{aligned}$$

where the absolute error is seen to be less than our error bounds computed above.

c. One of the roots of this function is $x = 0$. The Newton's method formula is given by

$$x_{n+1} = x_n - \frac{2e^{-2x} - 2\cos(2x) + 4\sin(x)}{-4e^{-2x} + 4\sin(2x) + 4\cos(x)}.$$

```
function z = newtonth1C(x0,tol,Nmax)
%NEWTON'S METHOD: Enter f(x), f'(x), x0, tol, Nmax
f = @(x) 2*exp(-2*x)-2*cos(2*x)+4*sin(x);
fp = @(x) -4*exp(-2*x)+4*sin(2*x)+4*cos(x);
xn = x0 - f(x0)/fp(x0);
z = [x0,xn];
y = f(xn);
i = 1;
fprintf('n = %d, x = %f, f(x) = %f\n', i, xn, y)
while (abs(xn - x0) ≥ tol)
   x0 = xn;
   xn = x0 - f(x0)/fp(x0);
   z = [z,xn];
   y = f(xn);
   i = i + 1;
   fprintf('n = %d, x = %f, f(x) = %f\n', i, xn, y)
      if (i ≥ Nmax)
        fprintf('Fail after %d iterations\n',Nmax);
```

```
            break
        end
end
end
```

The Newton iterations are

n	x_n	n	x_n	n	x_n
0	0.3000000	6	0.0040672	12	0.0000634
1	0.1380730	7	0.0020318	13	0.0000317
2	0.0668460	8	0.0010155	14	0.0000159
3	0.0329363	9	0.0005076	15	0.0000079
4	0.0163527	10	0.0002538		
5	0.0081482	11	0.0001269		

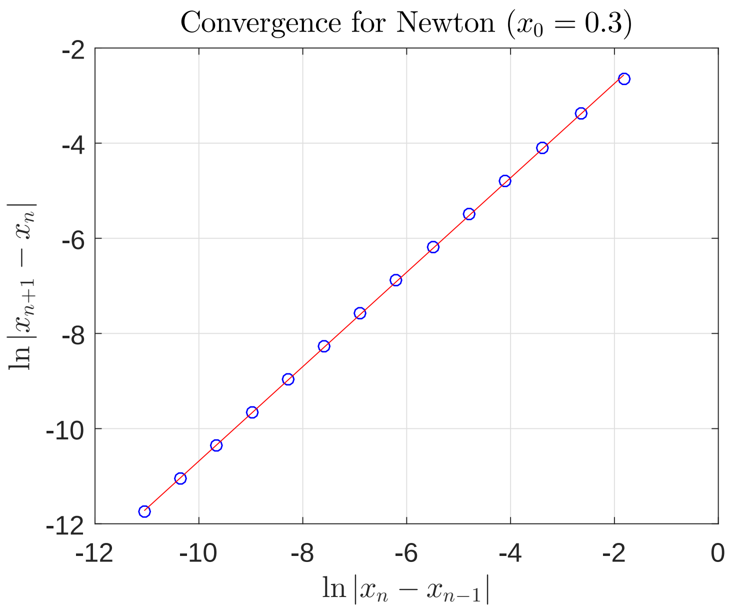
The rate of convergence is linear for the root at $x = 0$. It takes 15 iterations to converge to within 10^{-5}. The convergence is only linear because $x = 0$ is a double root. By examining the Cauchy sequence of the iteration,

$$\lim_{n\to\infty} \frac{|x_{n+1} - x_n|}{|x_n - x_{n-1}|^\alpha} = L,$$

becomes a line after taking logarithms:

$$\ln(|x_{n+1} - x_n|) = \ln(L) + \alpha \ln(|x_n - x_{n-1}|).$$

The linear least squares best fit from the table above gives $\alpha = 0.99202$ with $L = 0.46725$. Below is a graph of this convergence:



```
% convergence of Newton's method
z = newtonth1C(0.3,1e-5,20);
N = length(z);
```

```
X = log(abs(z(2:N-1)-z(1:N-2)));
Y = log(abs(z(3:N)-z(2:N-1)));
coef = polyfit(X,Y,1)
yL(1) = coef(1)*X(1)+coef(2);
yL(2) = coef(1)*X(N-2)+coef(2);
plot(X,Y,'bo');
hold on
plot([X(1),X(N-2)],[yL(1),yL(2)],'r-');grid;
title('Convergence for Newton ($x_0 = 0.3$)','FontSize',16,...
    'FontName','Times New Roman','interpreter','latex');
xlabel('$\ln|x_n - x_{n-1}|$','FontSize',16,'interpreter','latex');
ylabel('$\ln|x_{n+1} - x_n|$','FontSize',16,'interpreter','latex');
set(gca,'FontSize',14);
```

d. We use Halley's method to obtain quadratic convergence to the root at $x = 0$. This technique is given by

$$x_{n+1} = x_n - \frac{f(x)/f'(x)}{[f'(x)]^2 - f(x)f''(x)},$$

where $f'(x) = -4\,e^{-2x} + 4\,\sin(2x) + 4\,\cos(x)$ and $f''(x) = 8\,e^{-2x} + 8\,\cos(2x) - 4\,\sin(x)$. The MatLab program for this procedure is:

```
function z = halleyth1C(x0,tol,Nmax)
%NEWTON'S METHOD: Enter f(x), f'(x), x0, tol, Nmax
f = @(x) 2*exp(-2*x)-2*cos(2*x)+4*sin(x);
fp = @(x) -4*exp(-2*x)+4*sin(2*x)+4*cos(x);
ffp = @(x) 8*exp(-2*x)+8*cos(2*x)-4*sin(x);
xn = x0 - f(x0).*fp(x0)./(fp(x0).^2-f(x0).*ffp(x0));
z = [x0,xn];
y = f(xn);
i = 1;
fprintf('n = %d, x = %f, f(x) = %f\n', i, xn, y)
while (abs(xn - x0) ≥ tol)
   x0 = xn;
   xn = x0 - f(x0).*fp(x0)./(fp(x0).^2-f(x0).*ffp(x0));
   z = [z,xn];
   y = f(xn);
   i = i + 1;
   fprintf('n = %d, x = %f, f(x) = %f\n', i, xn, y)
      if (i ≥ Nmax)
        fprintf('Fail after %d iterations\n',Nmax);
        break
      end
end
end
```

This iteration to a tolerance of 10^{-8}, starting with $x_0 = 0.3$, is given by:

n	x_n
0	0.3
1	2.60295×10^{-2}
2	1.44384×10^{-4}
3	4.34370×10^{-9}
4	3.60007×10^{-9}

We can see that this method has quadratic convergence because the number of digits of accuracy for the solution roughly doubles with each iteration before the 4^{th} iterate. Alternately, if we examine

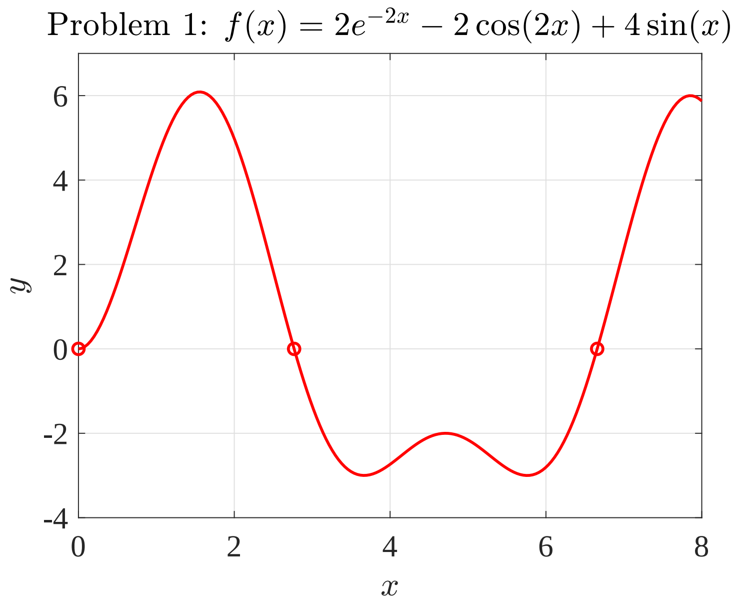
$$\frac{|x_{n+1} - x_*|}{|x_n - x_*|^2} = L,$$

for the first 3 iterates in the table with $x_* = 0$, we obtain $L \approx 0.25$, with $\alpha = 2$, giving quadratic convergence. Finally, we could use `polyfit` on these first 3 iterates with the formula:

$$\ln(|x_{n+1} - x_*|) = \alpha \ln(|x_{n+1} - x_*|) + \ln(L),$$

and obtain $\alpha = 2.03689$ and $L = 0.27712$, which agrees with the result above. However, the last iterate is only a slight improvement. This last iterate suffers from roundoff error in the subtractions, multiplications, and division. The calculations are too close to the digital limits of the computer.

e. There are two other roots of $f(x) = 0$ for $x \in (0, 8]$, which can be seen by the graph of the function.

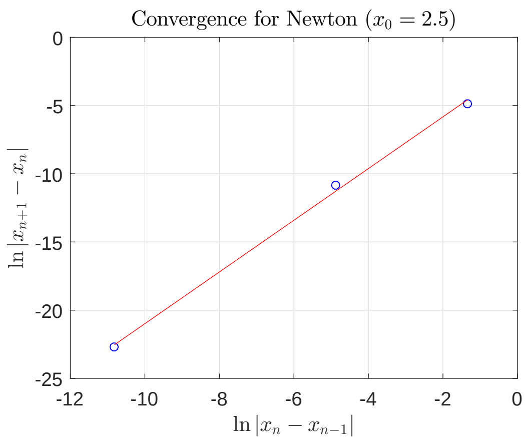


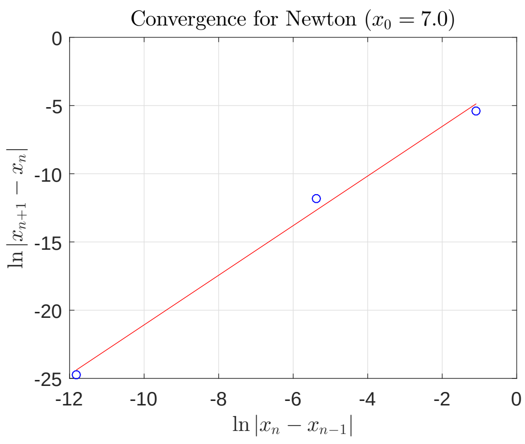
The two other roots are $x_1 = 2.76808143$ and $x_2 = 6.65791923$. The Newton's method near each of these roots converges quadratically and is given in the tables below. (This uses the same program above with different initial conditions.)

n	x_n
0	2.50000000
1	2.76036252
2	2.76806139
3	2.76808143
4	2.76808143

n	x_n
0	7.00000000
1	6.66258788
2	6.65792655
3	6.65791923
4	6.65791923

The table on the left shows the Newton iteration starting with $x_0 = 2.5$. Using the same Cauchy convergence as above, we obtain $\alpha = 1.89504$ and $L = 0.130254$, so this shows quadratic convergence. Below to the left is the graph for this convergence (using very similar program as seen in Part c).





The table on the right shows the Newton iteration starting with $x_0 = 7.0$. Using the same Cauchy convergence as above, we obtain $\alpha = 1.81836$ and $L = 0.0553891$, so this shows quadratic convergence. Above on the right is the graph for this convergence.

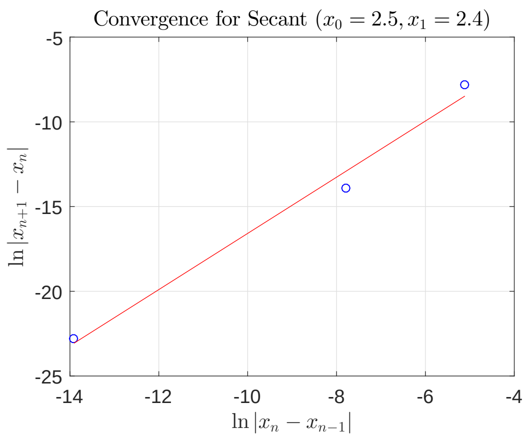
The secant finds the two roots are $x_1 = 2.76808143$ and $x_2 = 6.65791923$, like Newton's method. The secant method near each of these roots converges super linearly with $\alpha \approx 1.62$. The code for the secant method is given below, and it can be used to create the two tables below.

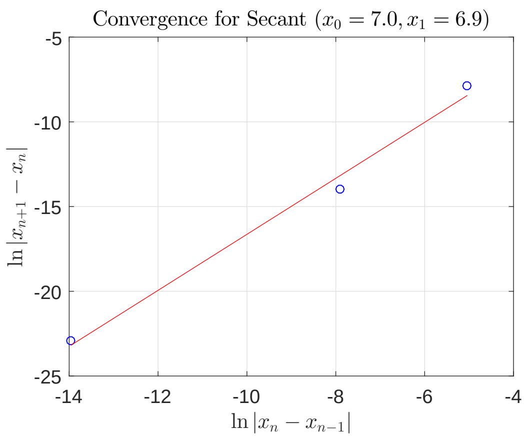
```
function z = secantth1C(x0,x1,tol,Nmax)
%SECANT METHOD: Enter f(x), x0, x1, tol, Nmax
f = @(x) 2*exp(-2*x)-2*cos(2*x)+4*sin(x);
xn = x1 - f(x1)*(x1-x0)/(f(x1)-f(x0));
z = [xn];
y = f(xn);
fprintf('x=%d, f(x) = %d\n', xn, y)
i = 1;
while (abs(xn - x1) ≥ tol)
 x0 = x1;
 x1 = xn;
 xn = x1 - f(x1)*(x0-x1)/(f(x0)-f(x1));
 z = [z,xn];
 y = f(xn);
 i = i + 1;
 if (i ≥ Nmax)
    fprintf('Fail after %d iterations\n',Nmax);
    break
 end

fprintf('x=%d, f(x) = %d\n', xn, y)
 end
 end
```

n	x_n
0	2.50000000
1	2.40000000
2	2.76163909
3	2.76766658
4	2.76808052
5	2.76808143
6	2.76808143

n	x_n
0	7.00000000
1	6.90000000
2	6.66476835
3	6.65829291
4	6.65792009
5	6.65791923
6	6.65791923

The table on the left shows the secant iteration starting with $x_0 = 2.5$ and $x_1 = 2.4$. Using the same Cauchy convergence as above, we obtain $\alpha = 1.65803$ and $L = 0.99625$, so this shows super linear convergence. Below to the left is the graph for this convergence (using very similar program as seen in Part c with the secant program replacing the Newton program).





The table on the right shows the secant iteration starting with $x_0 = 7.0$ and $x_1 = 6.9$. Using the same Cauchy convergence as above, we obtain $\alpha = 1.65279$ and $L = 0.89443$, so this shows super linear convergence. Above on the right is the graph for this convergence.

f. Basin of Attraction. A program is designed to test all starting values, $x_0 \in [0, 8]$ with a stepsize of 0.1. The programs are listed below.

```
function M = th1f(tol,Nmax)
%Basin of attraction
M = zeros(81,2);
x0 = 0;
for i = 1:81
    M(i,1) = x0;
    M(i,2) = th1fnewton_2(x0,tol,Nmax);
    x0 = x0 + 0.1;
end
end
```

```
function xn = th1fnewton_2(x0,tol,Nmax)
%NEWTON'S METHOD: Enter f(x), f'(x), x0, tol, Nmax
f = @(x) 2*exp(-2*x)-2*cos(2*x)+4*sin(x);
```

```
fp = @(x) -4*exp(-2*x)+4*sin(2*x)+4*cos(x);
xn = x0 - f(x0)/fp(x0);
i = 1;
while (abs(xn - x0) ≥ tol)
    x0 = xn;
    xn = x0 - f(x0)/fp(x0);
    i = i + 1;
    if (i ≥ Nmax)
        xn = 1000;
        break
    end
end
end
```

The table below list all the starting values, x_0, and shows what root the particular starting value converges to.

x_0	Root	x_0	Root	x_0	Root	x_0	Root
0	0	2.1	2.7681	4.1	6.6579	6.1	6.6579
0.1	0	2.2	2.7681	4.2	9.05004	6.2	6.6579
0.2	0	2.3	2.7681	4.3	6.6579	6.3	6.6579
0.3	0	2.4	2.7681	4.4	6.6579	6.4	6.6579
0.4	0	2.5	2.7681	4.5	6.6579	6.5	6.6579
0.5	0	2.6	2.7681	4.6	9.05004	6.6	6.6579
0.6	0	2.7	2.7681	4.7	38.0738	6.7	6.6579
0.7	0	2.8	2.7681	4.8	0	6.8	6.6579
0.8	0	2.9	2.7681	4.9	2.7681	6.9	6.6579
0.9	0	3	2.7681	5	2.7681	7	6.6579
1	0	3.1	2.7681	5.1	2.7681	7.1	6.6579
1.1	0	3.2	2.7681	5.2	0	7.2	6.6579
1.2	0	3.3	2.7681	5.3	0	7.3	6.6579
1.3	0	3.4	2.7681	5.4	50.6402	7.4	6.6579
1.4	0	3.5	0	5.5	2.7681	7.5	6.6579
1.5	0	3.6	0	5.6	2.7681	7.6	0
1.6	12.9411	3.7	19.2243	5.7	0	7.7	6.6579
1.7	0	3.8	9.05004	5.8	12.9411	7.8	0
1.8	6.6579	3.9	6.6579	5.9	9.05004	7.9	19.2243
1.9	2.7681	4	6.6579	6	0	8	9.05004
2	2.7681						

It is easy to see that any starting value $x_0 \in [0, 1.5]$ converges with Newton's method to the double root at $x_* = 0$. We also see that any $x_0 \in [1.9, 3.4]$ converges to the at $x_* = 2.7681$. Similarly, any $x_0 \in [6.1, 7.5]$ converges to the at $x_* = 6.6579$. These **3** intervals form the basins of attraction for their respective roots. There are **3** intervals, where the convergence to a root is less predictable, $[1.6, 1.9]$, $[3.5, 6]$, and $[7.6, 8]$. All of these intervals contain relative extrema for the function $f(x)$ as seen in the graph above. Hence, these intervals contain points of singularity for Newton's method, which shoots the tangent line as far away as 50 to find another root. No predictions can be easily made on these particular intervals.

2. Consider the discrete dynamical model given by

$$y_n = ay_{n-1} + by_{n-2}, \quad n \geq 2, \quad a, b \in \mathbb{R},$$

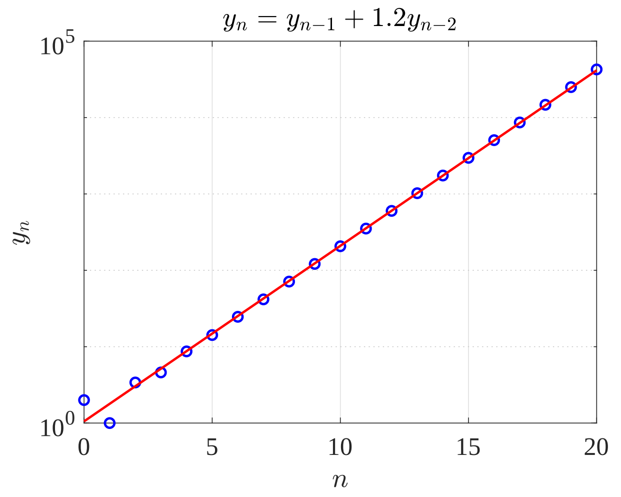
where a, b, y_0, y_1, and the maximum value of N are inputs. Below is a program that takes these parameters as input and produces the output `y`, which can be analyzed. In addition, a semi-log plot is created and is shown below.

```matlab
function y = dynsysC(a,b,y0,y1,N)
% dynamical system
y(1) = y0; y(2) = y1;
for i = 3:N+1
    y(i) = a*y(i-1) + b*y(i-2);
end
n = 0:N;
coefa = polyfit(n,log(y),1)
coef10a = polyfit(n,log10(y),1)
Y(1) = exp(coefa(2));
Y(2) = exp(20*coefa(1)+coefa(2));
semilogy(n,y,'bo');grid;
hold on
semilogy([0,20],[Y(1),Y(2)],'r-');
title('$y_n = y_{n-1} + 1.2y_{n-2}$','interpreter','latex');
xlabel('$n$','interpreter','latex');
ylabel('$y_n$','interpreter','latex');
set(gca,'FontSize',16);
hold off

print -depsc th3Ca_gr.eps
end
```

a. The first example begins with the parameters, $a = 1$, $b = 1.2$, $y_0 = 2$, $y_1 = 1$, and $N = 20$. The first 11 elements of the sequence are

$$y_n = [2, 1, 3.4, 4.6, 8.68, 14.2, 24.616, 41.656, 71.1952, 121.1824, 206.61664]$$

Graphing the first 20 elements gives the following graph:



If we apply the MatLab program `polyfit(n,Y,1)`, where `n = 0:20` and `Y = log(y(1:21))`, then we obtain the best fitting line is

$$\begin{aligned}\ln(y_n) &= 0.52916\,n + 0.051689,\\ \log_{10}(y_n) &= 0.22981\,n + 0.022448.\end{aligned}$$

Theoretically, we attempt a solution of the form $y_n = \lambda^n$, which when put into the dynamical model gives

$$\lambda^n = a\lambda^{n-1} + b\lambda^{n-2} \qquad \text{or} \qquad \lambda^{n-2}\left(\lambda^2 - a\lambda - b\right) = 0.$$

Thus, we have a **characteristic equation**, $\lambda^2 - a\lambda - b = 0$. For $a = 1$ and $b = 1.2$, we obtain the real solutions $\lambda_1 = -0.70416$ and $\lambda_2 = 1.70416$, which results in the general solution:

$$y_n = c_1(-0.70416)^n + c_2(1.70416)^n.$$

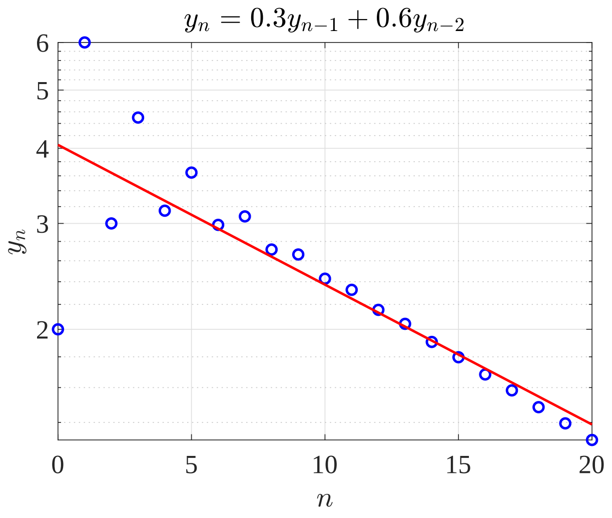
For large n, the first part of this solution is vanishingly small, so asymptotically, $y_n \approx c_2(1.70416)^n$, so $\ln(y_n) \approx \ln(c_2) + n\ln(1.70416)$. However, $\ln(1.70416) \approx .53307$, which is fairly close to the slope of the best fitting line above. The graph shows that the best fitting line closely fits the simulation.
From Maple, the actual solution to the difference equation is:

$$y_n = (1.70416)^n + (-0.70416)^n\,.$$

The second example begins with the parameters, $a = 0.3$, $b = 0.6$, $y_0 = 2$, $y_1 = 6$, and $N = 20$. The first 11 elements of the sequence are

$$y_n = [2, 6, 3, 4.5, 3.15, 3.645, 2.9835, 3.08205, 2.71472, 2.66364, 2.42792]$$

Graphing the first 20 elements gives the following graph:



Again we apply the MatLab program `polyfit(n,Y,1)`, where `n = 0:20` and `Y = log(y(1:21))`, then we obtain the best fitting line is

$$\begin{aligned}\ln(y_n) &= -0.0535498\,n + 1.399614,\\ \log_{10}(y_n) &= -0.0232564\,n + 0.607845.\end{aligned}$$

The characteristic equation for this model is $\lambda^2 - 0.3\lambda - 0.6 = 0$. It produces the real solutions $\lambda_1 = -0.638987$ and $\lambda_2 = 0.938987$, which results in the general solution:

$$y_n = c_1(-0.638987)^n + c_2(0.938987)^n.$$

For large n, the first part of this solution is vanishingly small, so asymptotically, $y_n \approx c_2(0.938987)^n$, so $\ln(y_n) \approx \ln(c_2) + n\ln(0.934847)$. However, $\ln(0.938987) \approx -0.0629540$, which is similar to the slope of the best fitting line above. The graph shows that the best fitting line moderately fits the simulation, but seems to have this slightly more negative slope.
From Maple, the actual solution to the difference equation is:

$$y_n = -2.61223\,(-0.638987)^n + 4.61223\,(0.938987)^n.$$

b. A code is created to add every third element for the discrete model:

$$y_n = y_{n-1} + 1.2\,y_{n-2}, \qquad y_0 = 2, \quad y_1 = 1,$$

provided the sum of the elements is less than $M = 1{,}000{,}000$.

```
function Y = dynsysCc(a,b,y0,y1,M)
% dynamical system
y(1) = y0; y(2) = y1;
sum = y0;
i = 2;
Y = [0, y0, sum];
while (sum < M)
    y(i+1) = a*y(i) + b*y(i-1);
    if (mod(i,3)==0)
        sum = sum + y(i+1);
        Y = [Y;i,y(i+1),sum];
    end
    i = i + 1;
end
j = i-4; % Index of last y(i) added
Sum = sum - y(i); % Sum subtracting last y added
fprintf('i = %d, sum = %f\n',j,Sum)

end
```

The output of this code generates a matrix containing the values of $3n$, y_{3n}, and $\sum_{n=0}^{N} y_{3n}$, including the first row with the sum exceeding M. For our problem where $M = 1{,}000{,}000$, we obtain the following table of values:

$3n$	y_{3n}	Sum
0	2	2
3	4.6	6.6
6	24.616	31.216
9	121.182	152.398
12	599.975	752.374
15	2969.290	3721.664
18	14695.494	18417.158
21	72730.205	91147.363
24	359952.754	451100.117

Our code requires iterations to y_{24}, and the sum satisfies:

$$\sum_{n=0}^{8} y_{3n} = 451,100.117 < 1,000,000.$$

We note that $y_{27} = 1,781,460.462$, which exceeds M.

3. The Greek mathematician Archimedes estimated the number π by approximating the circumference of a circle of diameter 1 by the perimeter of both inscribed and circumscribed polygons. The perimeter, t_n, of the circumscribed regular polygon with 2^n sides can be given by the recursive formula ($t_n > \pi$):

$$t_{n+1} = \frac{2^{n+1}\left(\sqrt{1+\left(\frac{t_n}{2^n}\right)^2}-1\right)}{\left(\frac{t_n}{2^n}\right)}, \qquad t_2 = 4.$$

a. Below is a MatLab program that satisfies the recursive relation above. It also includes the correction needed to produce a modified sequence that maintains accurate calculations.

```
function [T,Z] = circumpi(N)
%TH question x
n=2;
t(2) = 4; z(2) = 4;
T = zeros(N-1,1); Z = zeros(N-1,1);
T(1) = t(2);  Z(1) = z(2);
for n=3:N
    t(n) = (2^n)*(sqrt(1+(t(n-1)/2^(n-1))^2) - 1)/(t(n-1)/2^(n-1));
    z(n) = 2*z(n-1)/(sqrt(1+(z(n-1)/2^(n-1))^2) + 1);
    T(n-1) = t(n);
    Z(n-1) = z(n);
end
end
```

The problem in the first sequence is that the quantity in the numerator

$$\lim_{n\to\infty}\left(\sqrt{1+\left(\frac{t_n}{2^n}\right)^2}-1\right) = 0,$$

which has serious subtractive error. This is multiplied by a large number 2^{n+1} and divided by another number that tends to zero, $\left(\frac{t_n}{2^n}\right)$. Thus, we have multiple quantities that cause trouble in this calculation. The result is the NaN, which appears in the 30th iterate, terminating the sequence.

The program generates the following table:

n	t_n	t_n (modified)	n	t_n	t_n (modified)
2	4	4	17	3.14159252378847	3.14159265419139
3	3.31370849898476	3.31370849898476	18	3.14159086957911	3.14159265374019
4	3.18259787807452	3.18259787807452	19	3.14159252378847	3.14159265362739
5	3.15172490742925	3.15172490742925	20	3.14160058362633	3.14159265359919
6	3.14411838524586	3.1441183852459	21	3.14159252378847	3.14159265359214
7	3.14222362994234	3.14222362994245	22	3.1414451588708	3.14159265359038
8	3.1417503691697	3.14175036916896	23	3.14097079560248	3.14159265358994
9	3.14163208070224	3.14163208070318	24	3.13398329388535	3.14159265358983
10	3.14160251024197	3.14160251025681	25	3.11105678802532	3.1415926535898
11	3.14159511771836	3.14159511774959	26	3.05362474788829	3.14159265358979
12	3.14159326963174	3.1415932696293	27	2.61983729517921	3.14159265358979
13	3.14159280837968	3.14159280759964	28	3.05362474788829	3.14159265358979
14	3.14159269096296	3.14159269209225	29	0	3.14159265358979
15	3.14159267557045	3.1415926632154	30	NaN	3.14159265358979
16	3.14159269096296	3.14159265599619			

b. The key to algebraically making this problem tractable is multiplying the numerator by its conjugate

$$\left(\sqrt{1+\left(\frac{t_n}{2^n}\right)^2}+1\right)$$

and dividing by the same quantity. Thus, we have

$$\begin{aligned}
t_{n+1} &= \frac{2^{n+1}\left(\sqrt{1+\left(\frac{t_n}{2^n}\right)^2}-1\right)}{\left(\frac{t_n}{2^n}\right)}\frac{\left(\sqrt{1+\left(\frac{t_n}{2^n}\right)^2}+1\right)}{\left(\sqrt{1+\left(\frac{t_n}{2^n}\right)^2}+1\right)} \\
&= \frac{2^{n+1}\left(\frac{t_n}{2^n}\right)^2}{\left(\frac{t_n}{2^n}\right)\left(\sqrt{1+\left(\frac{t_n}{2^n}\right)^2}+1\right)} \\
&= \frac{2t_n}{\sqrt{1+\left(\frac{t_n}{2^n}\right)^2}+1}
\end{aligned}$$

This is the second sequence that appears in the program above and converges nicely to π.

4. To solve the quadratic equation

$$ax^2+bx+c=0,$$

we usually use the "classic" formula

$$x=\frac{-b\pm\sqrt{b^2-4ac}}{2a}. \tag{1}$$

This formula suffers from catastrophic subtraction errors on a finite-precision computer, whenever $b^2 \gg |4ac|$. This condition results in the sum or difference between b and $\sqrt{b^2-4ac}$ being almost 0, depending on the sign of b.

b. Assume a 4-digit computer (rounding last digit), and assume the quadratic equation has the parameters $a = 1.000$, $b = -71.82$, and $c = 0.2741$. We use the quadratic formula above to find the two real roots for this quadratic equation using the 4-digit computer.

We begin by computing $b^2 = 5158.112$, which the computer rounds to 5158. Next we compute $4ac = 1.0964$, which rounds to 1.096. The actual value of $b^2 - 4ac = 5157.016$, but the rounded calculation gives $b^2 - 4ac = 5157$. The actual value of $\sqrt{b^2 - 4ac} = 71.81237$, while the rounded value of $\sqrt{b^2 - 4ac} = 71.81$. The next calculations are $-b + \sqrt{b^2 - 4ac}$, which gives a precise value of 143.6324 and the rounded value of 143.6. The other calculations are $-b - \sqrt{b^2 - 4ac}$, which gives a precise value of 0.007633 and the rounded value of 0.01. Dividing by $2a$ gives the exact roots $r_1 = 71.816183$ and $r_2 = 0.003816688$. The rounded roots become $q_1 = 71.80$ and $r_2 = 0.005$. The percent errors are

$$100 \times \frac{(q_1 - r_1)}{r_1} = -0.0225\% \qquad \text{and} \qquad 100 \times \frac{(q_2 - r_2)}{r_2} = 31.0\%.$$

c. As we saw in the previous problem, we can use conjugates to eliminate the square root from the numerator. The quadratic formula gives

$$\begin{aligned} x &= \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \\ &= \frac{(-b \pm \sqrt{b^2 - 4ac})}{2a} \cdot \frac{(-b \mp \sqrt{b^2 - 4ac})}{(-b \mp \sqrt{b^2 - 4ac})} \\ &= \frac{4ac}{2a(-b \mp \sqrt{b^2 - 4ac})} \\ &= \frac{2c}{-b \mp \sqrt{b^2 - 4ac}}. \end{aligned}$$

Thus, this is an equivalent form of the quadratic formula.

Numerically, it is always better to avoid the subtractive error in either the numerator or the denominator. It follows that we have two cases:

1. If $b \geq 0$, then compute the two roots in the following manner:

$$r_1 = \frac{-b - \sqrt{b^2 - 4ac}}{2a} \qquad \text{and} \qquad r_2 = \frac{2c}{-b - \sqrt{b^2 - 4ac}}.$$

2. If $b < 0$, then compute the two roots in the following manner:

$$r_1 = \frac{2c}{-b + \sqrt{b^2 - 4ac}} \qquad \text{and} \qquad r_2 = \frac{-b + \sqrt{b^2 - 4ac}}{2a}.$$

Our example in Part b and d provides a good test case.

d. From the alternate formula, we find

$$q_1 = \frac{2c}{-b - \sqrt{b^2 - 4ac}} \qquad \text{and} \qquad q_2 = \frac{2c}{-b + \sqrt{b^2 - 4ac}}.$$

Most of the calculations are similar as performed in Part b. (The exact roots remain the same with $r_1 = 71.816183$ and $r_2 = 0.003816688$.) With the 4-digit rounding, we still obtain $-b - \sqrt{b^2 - 4ac} = 0.01$. We compute $2c = 0.5482$. Finally, we compute

$$q_1 = \frac{2c}{-b - \sqrt{b^2 - 4ac}} = \frac{0.5482}{0.01} = 54.82.$$

The percent error with this calculation is

$$100 \times \frac{(q_1 - r_1)}{r_1} = -23.67\%,$$

which is significantly worse. With the 4-digit rounding, we still obtain $-b + \sqrt{b^2 - 4ac} = 143.6$. We compute $2c = 0.5482$. Finally, we compute

$$q_2 = \frac{2c}{-b + \sqrt{b^2 - 4ac}} = \frac{0.5482}{143.6} = 0.003818.$$

The percent error with this calculation is

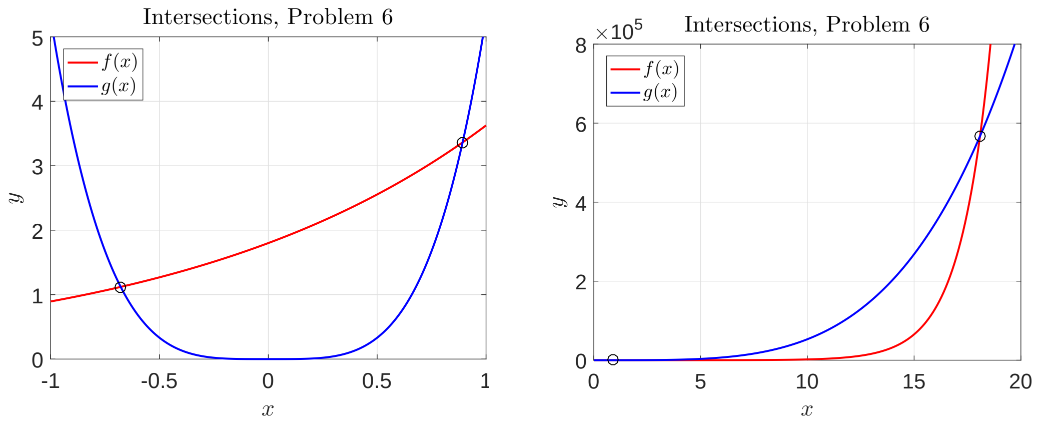
$$100 \times \frac{(q_2 - r_2)}{r_2} = 0.0344\%,$$

which is a significant improvement.

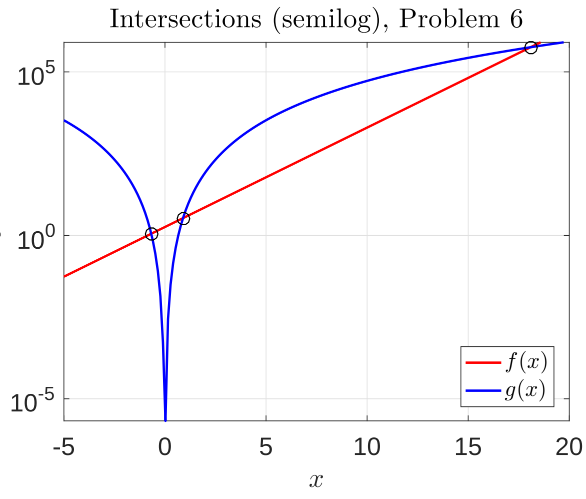
5. a. The functions

$$f(x) = 1.8\, e^{0.7x} \qquad \text{and} \qquad g(x) = 5.3\, x^4.$$

intersect at 3 points. We begin with graphing the functions to have an estimate of where the functions intersect. With normal axes, we divide into two graphs because of the different scales in y.



One can see all points of intersection if we use a semilog plot.



Below is one of the graphing programs.

```matlab
% Graphing script for Problem 6b
f = @(x) 1.8*exp(0.7*x);
g = @(x) 5.3*x.^4;
xy=linspace(0,20,200);
f2 = f(xy);
g2 = g(xy);

plot(xy,f2, 'r-','Linewidth',1.5);
hold on;
plot(xy,g2,'b-','Linewidth',1.5);
plot([0.892,18.087],[3.362,567153],'ko','Markersize',8);
grid;

xlim([0,20]);
ylim([0,800000]);

h = legend('$f(x)$', '$g(x)$', 'Location','northwest');
set(h,'Interpreter','latex')

fontlabs = 'Times New Roman';
xlabel('$x$','FontSize',16,'FontName',fontlabs,'interpreter','latex');
ylabel('$y$','FontSize',16,'FontName',fontlabs, 'interpreter','latex');
mytitle = 'Intersections, Problem 6';
title(mytitle,'FontSize',16,'FontName','Times New Roman','interpreter','latex');
set(gca,'FontSize',16);
hold off

print -depsc th6Cb_gr.eps
```

These curves intersect when $f(x) = g(x)$ or $F(x) = f(x) - g(x) = 0$, so we create programs for the bisection, secant, and Newton's methods to find all roots of $F(x) = 0$. We begin with a bisection program and run it to an accuracy of 10^{-5}. The initial points are based on the graphs above.

```matlab
function z = bisectionth6C(a,b,tol)
%BISECTION METHOD - Modify function below, then can
% find its roots in [a,b] to tolerance tol
f = @(x) 1.8*exp(0.7*x) - 5.3*x^4;
n = 0;
z = [(a+b)/2];
while (abs(b-a) ≥ tol)
    m = (a+b)/2;
    if (f(m) == 0)
        break;
    elseif(f(b)*f(m) < 0)
        a = m;
    else
        b = m;
    end
    y=f((a+b)/2);
    z = [z,(a+b)/2];
    n=n+1;
    fprintf('a = %f6, b= %f6, f=%f6,n=%d\n',a,b,y,n);
end
end
```

With the initial $a = -1$ and $b = 0$, the program is run with the command `z = bisectionth6C(-1,0,1e-5)`, which yields the root $r_1 = -0.67798996$ after 18 iterations. With the initial $a = 0$ and $b = 1$, the program is run with the command `z = bisectionth6C(1,2,1e-5)`, which yields the root $r_2 = 0.89242935$ after 18 iterations. Finally with the initial $a = 13$ and $b = 14$, the program is run with the command `z = bisectionth6C(15,20,1e-5)`, which yields the root $r_3 = 18.08657169$ after 20 iterations.

Before moving to the secant method, we explore the convergence of the bisection method. Using other information (such as running convergence of Newton's method to a high tolerance), we find the negative root to be $x_* = -0.6779866372$ (accurate to 10 significant figures). The order of convergence is found by:

$$\lim_{n\to\infty} \frac{|x_{n+1} - x_*|}{|x_n - x_*|^\alpha} = \lambda.$$

We rearrange this to examine

$$\ln(|x_{n+1} - x_*|) = \alpha \ln(|x_n - x_*|) + \ln(\lambda), \tag{2}$$

where α is the order of convergence. Our program above produces the bisection iterates, so we can readily compare them to x_*. By making a log-log plot and finding a linear least squares fit, the best values of α and λ can be found. Alternately, we can examine Cauchy convergence with the formula:

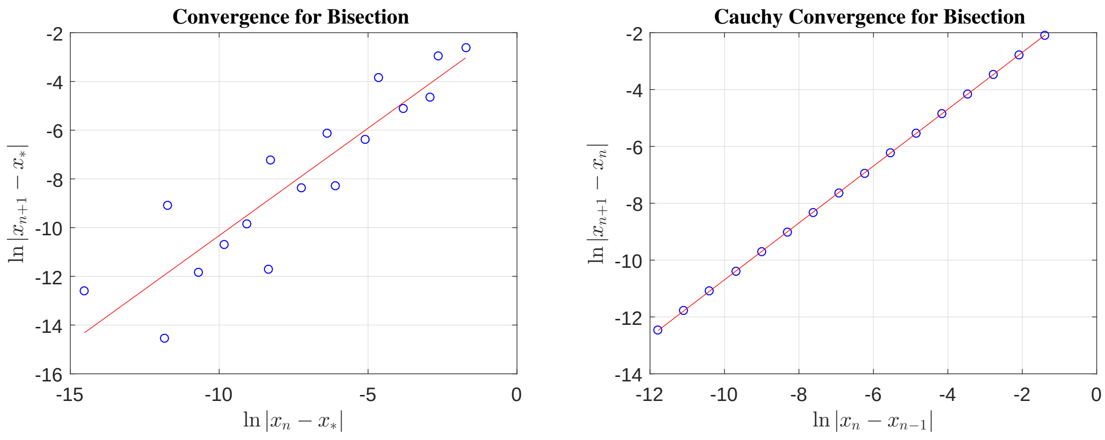
$$\lim_{n\to\infty} \frac{|x_{n+1} - x_n|}{|x_n - x_{n-1}|^\alpha} = \lambda.$$

We run the MatLab program, using `[coef,coef2] = bisect_conv_6C(-1,0,1e-5,-0.6779866372)`.

```
function [coef,coef2] = bisect_conv_6C(a,b,tol,xs)
% convergence of bisection method
z = bisectionth6C(a,b,tol);
N = length(z);
X = log(abs(z(1:N-1)-xs));
Y = log(abs(z(2:N)-xs));
coef = polyfit(X,Y,1);
yL(1) = coef(1)*X(1)+coef(2);
yL(2) = coef(1)*X(N-1)+coef(2);
plot(X,Y,'bo');
hold on
plot([X(1),X(N-1)],[yL(1),yL(2)],'r-');grid;
title('Convergence for Bisection','FontSize',16,'FontName','Times New Roman');
xlabel('$\ln|x_n - x_*|$','FontSize',16,'interpreter','latex');
ylabel('$\ln|x_{n+1} - x_*|$','FontSize',16,'interpreter','latex');
set(gca,'FontSize',14);
hold off
print -depsc th6Cca1_gr.eps

figure(102)
```

```
X2 = log(abs(z(2:N-1)-z(1:N-2)));
Y2 = log(abs(z(3:N)-z(2:N-1)));
coef2 = polyfit(X2,Y2,1);
yL2(1) = coef2(1)*X2(1)+coef2(2);
yL2(2) = coef2(1)*X2(N-2)+coef2(2);
plot(X2,Y2,'bo');
hold on
plot([X2(1),X2(N-2)],[yL2(1),yL2(2)],'r-');grid;
title('Cauchy Convergence for Bisection','FontSize',16,'FontName','Times New ...
    Roman');
xlabel('$\ln|x_n - x_{n-1}|$','FontSize',16,'interpreter','latex');
ylabel('$\ln|x_{n+1} - x_n|$','FontSize',16,'interpreter','latex');
set(gca,'FontSize',14);
hold off
print -depsc th6Cca2_gr.eps
end
```

This code produces the graphs below with the standard definition of convergence on the left and the Cauchy convergence on the right.



The coefficients for the standard convergence are:

$$\alpha = 0.881605 \qquad \text{and} \qquad \lambda = 0.220729,$$

while the Cauchy convergence (with no surprise) produces:

$$\alpha = 1.000 \qquad \text{and} \qquad \lambda = 0.500.$$

These calculations confirm what we expected that the bisection method is **linear** with $\alpha \approx 1$.

We next consider the secant method and will run it to an accuracy of 10^{-8}. The initial points are based on the graphs above.

```
function z = secantthC6(x0,x1,tol,Nmax)
f = @(x) 1.8*exp(0.7*x) - 5.3*x^4;
xn = x1 - f(x1)*(x1-x0)/(f(x1)-f(x0));
z = [xn];
y = f(xn);
```

```
fprintf('x=%d, f(x) = %d\n', xn, y)
i = 1;
while (abs(xn - x1) ≥ tol)
 x0 = x1;
 x1 = xn;
 xn = x1 - f(x1)*(x0-x1)/(f(x0)-f(x1));
 z = [z,xn];
 y = f(xn);
 i = i + 1;
 if (i ≥ Nmax)
    fprintf('Fail after %d iterations\n',Nmax);
    break
 end

fprintf('x=%d, f(x) = %d\n', xn, y)
 end
 end
```

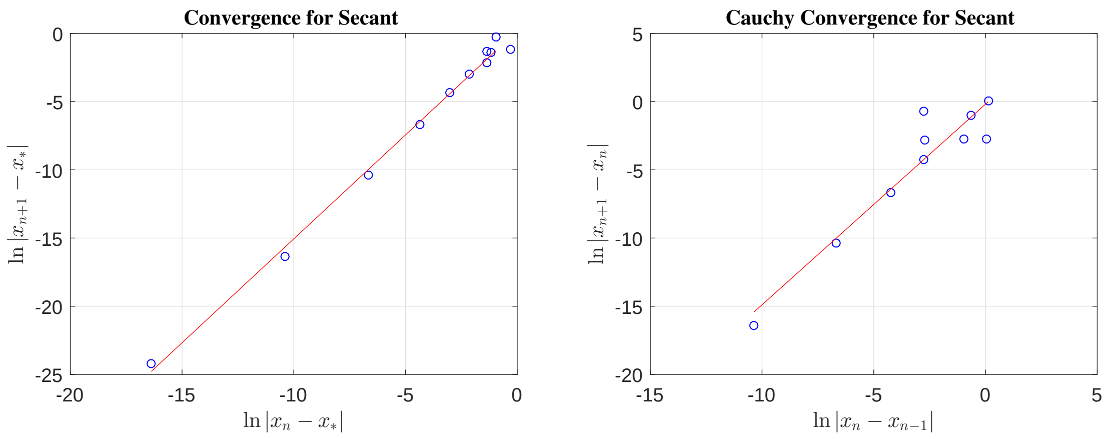
With the initial $x_0 = -1$ and $x_1 = 0$, the program is run with the command `z = secantthC6(-1,0,1e-8,20)`, which yields the root $r_1 = -0.6779866372$ after 13 iterations. With the initial $x_0 = 0$ and $x_1 = 1$, the program is run with the command `z = secantthC6(0,1,1e-8,20)`, which yields the root $r_2 = 0.8924325497$ after 9 iterations. Finally with the initial $x_0 = 13$ and $x_1 = 14$, the program is run with the command `z = secantthC6(15,20,1e-8,30)`, which yields the root $r_3 = 16.2759908550$ after 5 iterations.

We explore the convergence of the secant method. Again we use the negative root as $x_* = -0.6779866372$ (accurate to 10 significant figures). The order of convergence is studied with (2), where α is the order of convergence, and the variant form of the Cauchy convergence. Our program above produces the secant iterates, so we can readily compare them to x_* or in the Cauchy sense. By making a log-log plot and finding a linear least squares fit, the best values of α and λ can be found. We run the MatLab program, using `[coef,coef2] = ... secant_conv_6C(-1,0,1e-8,-0.6779866372,20)`.

```
function [coef,coef2] = secant_conv_6C(a,b,tol,xs,Nmax)
% convergence of secant method
z = secantthC6(a,b,tol,Nmax);
N = length(z)-1;
X = log(abs(z(1:N-1)-xs));
Y = log(abs(z(2:N)-xs));
coef = polyfit(X,Y,1);
yL(1) = coef(1)*X(1)+coef(2);
yL(2) = coef(1)*X(N-1)+coef(2);
plot(X,Y,'bo');
hold on
plot([X(1),X(N-1)],[yL(1),yL(2)],'r-');grid;
title('Convergence for Secant','FontSize',16,'FontName','Times New Roman');
xlabel('$\ln|x_n - x_*|$','FontSize',16,'interpreter','latex');
ylabel('$\ln|x_{n+1} - x_*|$','FontSize',16,'interpreter','latex');
set(gca,'FontSize',14);
hold off
print -depsc th6Ccb1_gr.eps

figure(102)
```

```
X2 = log(abs(z(2:N-1)-z(1:N-2)));
Y2 = log(abs(z(3:N)-z(2:N-1)));
coef2 = polyfit(X2,Y2,1);
yL2(1) = coef2(1)*X2(1)+coef2(2);
yL2(2) = coef2(1)*X2(N-2)+coef2(2);
plot(X2,Y2,'bo');
hold on
plot([X2(1),X2(N-2)],[yL2(1),yL2(2)],'r-');grid;
title('Cauchy Convergence for Secant','FontSize',16,'FontName','Times New ...
    Roman');
xlabel('$\ln|x_n - x_{n-1}|$','FontSize',16,'interpreter','latex');
ylabel('$\ln|x_{n+1} - x_n|$','FontSize',16,'interpreter','latex');
set(gca,'FontSize',14);
hold off
print -depsc th6Ccb2_gr.eps
end
```

This code produces the graphs below with the standard definition of convergence on the left and the Cauchy convergence on the right.



The coefficients for the standard convergence are:

$$\alpha = 1.523647 \qquad \text{and} \qquad \lambda = 1.190484,$$

while the Cauchy convergence produces:

$$\alpha = 1.470615 \qquad \text{and} \qquad \lambda = 0.828452.$$

These calculations give slightly lower convergence rates to what we expected, but still show that the secant method is **super linear** with $\alpha \approx 1.62$.

Finally, we explore Newton's method and will run it to an accuracy of 10^{-8}. The initial points are based on the graphs above.

```
function z = newtonth6C(x0,tol,Nmax)
%NEWTON'S METHOD: Enter f(x), f'(x), x0, tol, Nmax
f = @(x) 1.8*exp(0.7*x) - 5.3*x^4;
fp = @(x) 1.26*exp(0.7*x) - 21.2*x^3;
```

```
xn = x0 - f(x0)/fp(x0);
z = [x0,xn];
y = f(xn);
i = 1;
fprintf('n = %d, x = %f, f(x) = %f\n', i, xn, y)
while (abs(xn - x0) ≥ tol)
   x0 = xn;
   xn = x0 - f(x0)/fp(x0);
   z = [z,xn];
   y = f(xn);
   i = i + 1;
   fprintf('n = %d, x = %f, f(x) = %f\n', i, xn, y)
      if (i ≥ Nmax)
        fprintf('Fail after %d iterations\n',Nmax);
        break
      end
end
end
```

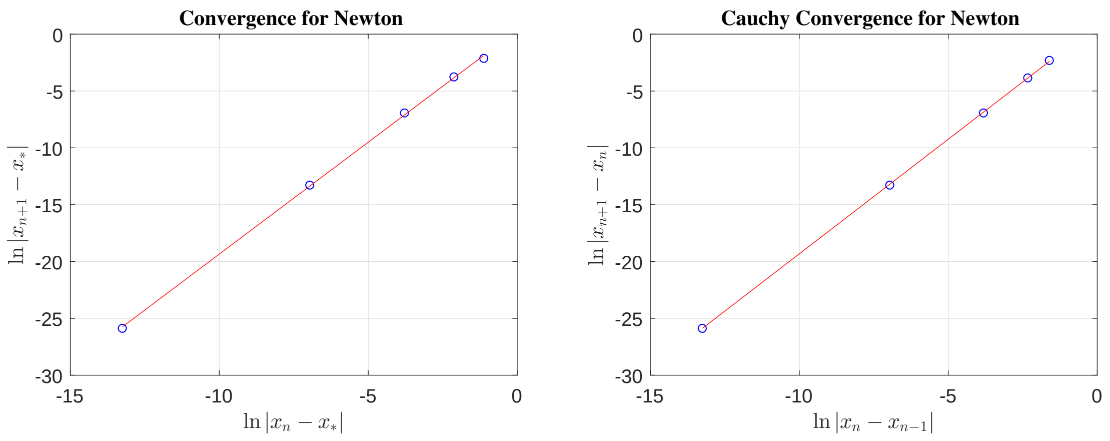
With the initial $x_0 = -1$, the program is run with the command `z = newtonth6C(-1,1e-8,20)`, which yields the root $r_1 = -0.6779866372$ after 7 iterations. With the initial $x_0 = 1$, the program is run with the command `z = newtonth6C(1,1e-8,20)`, which yields the root $r_2 = 0.8924325497$ after 6 iterations. Finally with the initial $x_0 = 13$, the program is run with the command `z = newtonth6C(20,1e-8,20)`, which yields the root $r_3 = 18.0865696895$ after 8 iterations. This program clearly requires the fewest number of iterations.

We explore the convergence of Newton's method. Again we use the second root as $x_* = -0.677986637173649$ (accurate to 15 significant figures). The order of convergence is studied with (2) or the Cauchy convergence, where α is the order of convergence. Our program above produces the Newton iterates, so we can readily compare them to x_* or in the Cauchy sense. By making a log-log plot and finding a linear least squares fit, the best values of α and λ can be found. (We had to drop the last iterate as it was too close to x_* and dominated by round-off error.) We run the MatLab program, using `[coef,coef2] = ...` `newton_conv_6C(-1,1e-8,-0.677986637173649,20)`.

```
function [coef,coef2] = newton_conv_6C(x0,tol,xs,Nmax)
% convergence of Newton's method
z = newtonth6C(x0,tol,Nmax);
N = length(z);
X = log(abs(z(1:N-2)-xs));
Y = log(abs(z(2:N-1)-xs));
coef = polyfit(X,Y,1);
yL(1) = coef(1)*X(1)+coef(2);
yL(2) = coef(1)*X(N-2)+coef(2);
plot(X,Y,'bo');
hold on
plot([X(1),X(N-2)],[yL(1),yL(2)],'r-');grid;
title('Convergence for Newton','FontSize',16,'FontName','Times New Roman');
xlabel('$\ln|x_n - x_*|$','FontSize',16,'interpreter','latex');
ylabel('$\ln|x_{n+1} - x_*|$','FontSize',16,'interpreter','latex');
set(gca,'FontSize',14);
hold off
print -depsc th6Ccc1_gr.eps
```

```

figure(102)
X2 = log(abs(z(2:N-1)-z(1:N-2)));
Y2 = log(abs(z(3:N)-z(2:N-1)));
coef2 = polyfit(X2,Y2,1);
yL2(1) = coef2(1)*X2(1)+coef2(2);
yL2(2) = coef2(1)*X2(N-2)+coef2(2);
plot(X2,Y2,'bo');
hold on
plot([X2(1),X2(N-2)],[yL2(1),yL2(2)],'r-');grid;
title('Cauchy Convergence for Newton','FontSize',16,'FontName','Times New ...
    Roman');
xlabel('$\ln|x_n - x_{n-1}|$','FontSize',16,'interpreter','latex');
ylabel('$\ln|x_{n+1} - x_n|$','FontSize',16,'interpreter','latex');
set(gca,'FontSize',14);
hold off
print -depsc th6Ccc2_gr.eps
end
```

This code produces the graphs below with the standard definition of convergence on the left and the Cauchy convergence on the right.



The coefficients for the standard convergence are:

$$\alpha = 1.968593 \qquad \text{and} \qquad \lambda = 1.401414,$$

while the Cauchy convergence produces:

$$\alpha = 2.0159258593 \qquad \text{and} \qquad \lambda = 2.305236.$$

These calculations give the convergence rates to what we expected, showing that Newton's method is **Quadratic** with $\alpha = 2$.

These results found the x-values of the intersection of $f(x)$ and $g(x)$. When these values are inserted into $f(x)$ or $g(x)$, we obtain the y-values. The table below gives the three points of intersection.

Points of intersection

$x_1 = -0.67798664,$	$y_1 = 1.11985142$
$x_2 = 0.89243255,$	$y_2 = 3.36184338$
$x_3 = 18.08656969,$	$y_3 = 567,153.5996$