Note: Each of the approximations was calculated by editing the posted code online. Namely, only the necessary functions, f, fprime, etc. were edited as necessary for the specific problem.

Section 2.1 – 4

a) [-2, -1], Bisection terminates in 7 iterations returning the approximation -1.4140625.
b) [0, 2], Bisection terminates in 8 iterations returning the approximation 1.4140625.
b) [2, 3], Bisection terminates in 7 iterations returning the approximation 2.7265625.
b) [-1, 0], Bisection terminates in 7 iterations returning the approximation -0.7265625.

Section 2.1 – 12

This is the same as finding a root to the equation \(x^2 - 3 = 0\). We know that the root is in between 1 and 2. Setting the tolerance to \(10^{-4}\), we get the approximation of the square root of three to be 1.731994629. Using a calculator, we get 1.732050808, so the error is about \(5.6 \times 10^{-5}\), within the desired range.

Section 2.1 – 14

We need to find the minimum value of n for which \(\frac{4-1}{2^n} < 0.001\). Solving this, we get:

\[
2^n > 3000
\]

So, \(n > \log_2 3000 \approx 11.55\). Since n must be integral, we need 12 iterations.

Use Theorem 2.1 to find a bound for the number of iterations needed to achieve an approximation with accuracy \(10^{-3}\) to the solution of \(x^3 + x - 4 = 0\) lying in the interval [1,4]. Find an approximation of the root with this degree of accuracy.

Running bisection.c with the appropriate function for 12 iterations yielded the approximation for the root to be 1.378662109.

Section 2.2 – 4

a) Diverges very quickly. \(p_1 = 343\) and \(p_2\) has 27 digits.

b) Diverges pretty quickly as well. \(p_1 = 7\) and \(p_2 \approx -335\) and then it’s off the to races.

c) Converges to within \(10^{-10}\) in 8 iterations. The approximate value of the root is 1.475773162. This is essentially Newton’s Method.
d) Converges to within $10^{-2}$ in 67 iterations. This method is fairly haphazard bouncing back and forth below and above the root. 837 iterations were needed to get within $10^{-10}$ of the actual root. This is many more iterations than the function in part c.

**Section 2.2 – 6**

Use a fixed-point iteration method to determine a solution accurate to within $10^{-2}$ for $x^3 - x - 1 = 0$ on $[1, 2]$. Use $p_0 = 1$.

Rewrite the function as $x = f(x)$ such that $f'(x)$ on $[1,2]$ is in between -1 and 1, exclusive.

One possibility is as follows:

for $x^3 = x + 1$, so $x = \sqrt[3]{x + 1}$.

Using the fixed point algorithm with this representation of the function with $p_0 = 1$, we get an approximation of the root of 1.324717957 in 15 iterations. It converges to $10^{-2}$ of the real answer in 3 iterations. ($p_3 = 1.322353819$.)

**Section 2.3 – 2**

Using Newton's method, with $f(x) = -x^3 - \cos x$ and $p_0 = -1$, $p_2 = -0.865684163$. The method converges to -0.865474033 in 5 iterations.

My program online converges in to within $10^{-9}$ of the real answer in 21 iterations because of the following line:

```java
if (fprime(p0) == 0)
    p0 += .01;
```

Without this hack, the code would create a divide by zero error, since for this particular function $f'(0) = 0$.

**Section 2.3 – 24**

\[
\begin{align*}
15640000 &= 10000000e^\lambda + \frac{435000}{\lambda}(e^\lambda - 1) \\
1564000\lambda &= 10000000\lambda e^\lambda + 435000(e^\lambda - 1) \\
1564000\lambda + 435000 &= 10000000\lambda e^\lambda + 435000e^\lambda \\
1564000\lambda + 435000 &= (1000000\lambda + 435000)e^\lambda \\
1564000\lambda + 435000 &= e^\lambda \\
\frac{1000000\lambda + 435000}{1564000\lambda + 435000} &= e^\lambda \\
\lambda &= \ln\left(\frac{1564000\lambda + 435000}{1000000\lambda + 435000}\right)
\end{align*}
\]
I probably should have used Newton here, but I decided to use the fixed point algorithm, so I didn’t have to take the derivative of that natural log function, with respect to lambda. The fixed point algorithm took 75 steps to converge to the answer lambda = 0.100997935. If we just insist on accuracy to $10^{-4}$, using fixed point, after 22 iterations, we get 0.10102.

Now, we can use this value to plug into the following equation on page 47 of the text:

\[ N(t) = 1000000e^{\lambda t} + \frac{435000}{\lambda}(e^{\lambda t} - 1) \]

Plugging in $\lambda = 0.100997935$ and $t = 2$, we get the following:

\[ N(2) = 1000000e^{2\lambda} + \frac{435000}{\lambda}(e^{2\lambda} - 1) \approx 2187938 \]

Note: If we plug in $\lambda = 0.1010$, which is correct to 4 significant digits, we get 2187945 for the estimate.

**Section 2.3 – 26**

Using the information in the question, we want $A = 750000$, $P = 1500$, and $n = 240$ (monthly periods). This gives us the following equation:

\[ 750000 = \frac{1500}{i}[ (1 + i)^{240} - 1 ] \]

\[ 750000i = 1500[(1 + i)^{240} - 1] \]

\[ 500i = (1 + i)^{240} - 1 \]

\[ 500i + 1 = (1 + i)^{240} \]

\[ \ln (500i + 1) = \ln(1 + i)^{240} \]

\[ \ln (500i + 1) = 240\ln(1 + i) \]

\[ \ln (i + 1) = \frac{240}{\ln (500i + 1)} \]

\[ i + 1 = e^{\left(\frac{\ln(500i+1)}{240}\right)} \]

\[ i = e^{\left(\frac{\ln(500i+1)}{240}\right)} - 1 \]

Once again, this is naturally set up to use the fixed point algorithm. Though Newton would be faster, I am pretty sure fixed-point will work and I’ll avoid taking the derivative of this function. (Though, the derivative of this function is very, very easy…)

After running the algorithm for 31 iterations, we arrive at $i = 0.005550782$. This is a monthly interest rate. The corresponding yearly interest rate is $(1+i)^{12} - 1 = 0.0686810218$. But, often times, when someone quotes an APR that is applied by compounding monthly, what actually happens is that the rate is divided by 12 and that rate is applied each month instead. Depending
on your interpretation, one gets different annual percentage rates. If we multiply the value above by 12, we get 6.67%, which agrees with the textbook.

**Note:** I decided to test this value in python and found it to work fairly well:

```python
sum = 0
for i in range(240):
    sum = sum*(1.005550782)
    sum = sum + 1500
print(sum)
```

I got sum to print out as $750000.01, which is very close.

**Section 2.4 – 2a**

\[
\begin{align*}
    f(x) &= 1 - 4x\cos(x) + 2x^2 + \cos(2x) \\
    f'(x) &= -4(-x\sin(x) + \cos(x)) + 4x - 2\sin(2x) \\
    f''(x) &= 4x\sin(x) - 4\cos(x) + 4x - 2\sin(2x)
\end{align*}
\]

Now, plug these functions into Newton’s Method. Since we can’t evaluate the derivative at \( x = 0 \), choose a different point, say, \( x = 0.5 \) to evaluate the function.

After 73 iterations, Newton’s method yields \( x = 0.739085130 \) as the root of the equation. If we run it until we get to an accuracy of \( 10^{-5} \), we get 0.739078679 after 15 iterations.

**Section 2.4 – 2b**

\[
\begin{align*}
    f(x) &= x^6 + 6x^5 + 9x^4 - 2x^3 - 6x^2 + 1 \\
    f'(x) &= 6x^5 + 30x^4 + 36x^3 - 6x^2 - 12x \\
    f''(x) &= 30x^4 + 120x^3 + 108x^2 - 12x - 12
\end{align*}
\]

Using Newton’s Method, adjusted to use the second derivative as well in each iteration, starting the algorithm with the starting point of -3, it took 4 iterations to arrive at the root \( x = -2.879385 \).

**Note:** This code is the added example to the course web page, adjnewtonmethod.c.