Problem 1

5.18 The saturation concentration of dissolved oxygen in freshwater can be calculated with the equation (APHA, 1992)

$$\ln o_{sf} = -139.34411 + \frac{1.575701 \times 10^5}{T_a} - \frac{6.642308 \times 10^7}{T_a^2} + \frac{1.243800 \times 10^{10}}{T_a^3} - \frac{8.621949 \times 10^{11}}{T_a^4}$$

where o_{sf} = the saturation concentration of dissolved oxygen in freshwater at 1 atm (mg/L) and T_a = absolute temperature (K). Remember that $T_a = T + 273.15$, where T = temperature (°C). According to this equation, saturation decreases with increasing temperature. For typical natural waters in temperate climates, the equation can be used to determine that oxygen concentration ranges from 14.621 mg/L at 0°C to 6.413 mg/L at 40°C. Given a value of oxygen concentration, this formula and the bisection method can be used to solve for temperature in °C.

(a) If the initial guesses are set as 0 and 40°C, how many bisection iterations would be required to determine temperature to an absolute error of 0.05°C?

(b) Develop and test a bisection program to determine T as a function of a given oxygen concentration to a prespecified absolute error as in **(a)**. Given initial guesses of 0 and 40°C, test your program for an absolute error = 0.05°C and the following cases: o_{sf} = 8, 10 and 12 mg/L. Check your results.

[1] - Solve problem 5.18(b) in the textbook as follows:

Produce code to implement the bisection method using your favorite programming language (VBA/Excel, C++, Matlab, Etc.). Your code should consists of a main program (say, *MainBisect*), and a function (*Bisect*):

The main program should do the following:

(1) input the following values:

- *xl*, the lowest value in a bracket for the bisection method
- *xu*, the upper value in a bracket for the bisection method
- *Ead*, the desired absolute error
- *imax,* the maximum allowed number of iterations before declaring a divergent process
- *osf*, the variable o_{sf} given for problem 5.18 (b) – see more details below

(2) check if *xl* and *xu* are suitable for the bisection method, i.e., check that *f*(*xl*)**f*(*xu*) < 0. If not, the program should report this situation as an error and stop.
(3) call function or subroutine *Bisect* (described below) to return a solution
(4) outputs the solution returned by *Bisect*, the number of iterations required to solve the equation, and the absolute error at the point that the solution was found, and the value of the function *f*(*x*) at the solution point (i.e., the values *xr*, *iter*, *Ea*, from *Bisect*, and *f*(*xr*)).

```
Problem 1:
          Estimating Roots Between 0-40
osf       root           loops     error
8         26.7871        12        0.0364564
10        15.3857        13        0.031736
12        7.46338        14        0.0327118
---------------------------------------------------
```

Solution:

assign2.cpp:

```
//----------------------------------------------------------------------------
------+
// Estimate Bisect
|
// Given an arbitrary polynomial, use the bisection method to solve
|
// The equation "f(root) = 0", assuming l_bound < root < r_bound
|
//----------------------------------------------------------------------------
------+
Estimate assign2:: Bisect(
                std::function<double(double)> f,
                double l_bound,
                double r_bound,
                double max_loops,
                double max_error) {

    Estimate est(l_bound);
    double lSign = f(l_bound);

    if (lSign * f(r_bound) >= 0) {
        std::cout <<
            "Error: left-hand and right-hand estimates do not bound a root"
<< std::endl;
        return est;
    }
```

```cpp
    double prev, sign;
    do {
        ++est.loops;
        prev = est.value;
        est.value = (l_bound + r_bound) / 2;

        if (est.value != 0) {
            est.error = abs(est.value - prev) * 100 / est.value;
        }

        sign = f(est.value) * lSign;
        if (sign < 0) {
            r_bound = est.value;
        } else if (sign > 0) {
            l_bound = est.value;
        } else {
            est.error = 0;
        }
    } while (est.loops < max_loops && est.error >= max_error);

    return est;
}

//---------------------------------------------------------------------------
------+
// Problem 1 - Using the Bisection Method
|
//---------------------------------------------------------------------------
------+
void assign2::Problem1() {
    Estimate est;
    double osf = 8;
    auto polynomial = [&osf](double x) -> double {
        double T_Kelvin[5];
        Powers(T_Kelvin, 5, x + 273.15);

        return log(osf)
            + 139.34411
            - (157570.1 / T_Kelvin[1])
            + (66423080 / T_Kelvin[2])
            - (12438000000 / T_Kelvin[3])
            + (862194900000 / T_Kelvin[4]);
    };

    const int
        TITLE = 50,
        COL_OSF = 10,
        COL_ROOT = 15,
        COL_LOOPS = 10,
        COL_ERROR = 15;

    std::cout << "Problem 1:" << std::endl
        << std::setw(TITLE) << centered("Estimating Roots Between 0-40")
        << std::endl << std::left
        << std::setw(COL_OSF) << "osf"
        << std::setw(COL_ROOT) << "root"
```

```
        << std::setw(COL_LOOPS) << "loops"
        << std::setw(COL_ERROR) << "error"
        << std::endl;

    est = Bisect(polynomial, 0, 40, 100, 0.05);
    std::cout
        << std::setw(COL_OSF) << "8"
        << std::setw(COL_ROOT) << est.value
        << std::setw(COL_LOOPS) << est.loops
        << std::setw(COL_ERROR) << est.error
        << std::endl;

    osf = 10;
    est = Bisect(polynomial, 0, 40, 100, 0.05);
    std::cout
        << std::setw(COL_OSF) << "10"
        << std::setw(COL_ROOT) << est.value
        << std::setw(COL_LOOPS) << est.loops
        << std::setw(COL_ERROR) << est.error
        << std::endl;

    osf = 12;
    est = Bisect(polynomial, 0, 40, 100, 0.05);
    std::cout
        << std::setw(COL_OSF) << "12"
        << std::setw(COL_ROOT) << est.value
        << std::setw(COL_LOOPS) << est.loops
        << std::setw(COL_ERROR) << est.error
        << std::endl;

    std::cout << "---------------------------------------------------" <<
std::endl << std::endl;
}
```

assign2.h:

```
#pragma once
#include <functional>

namespace assign2 {
    struct Estimate {
        double error;
        double value;
        int loops;
        Estimate(double est = 0) {
            error = 100;
            loops = 0;
            value = est;
        }
    };

    Estimate Bisect(std::function<double(double)>, double, double, double,
double);
    void Problem1();
}
```

Problem 2

5.22 Many fields of engineering require accurate population estimates. For example, transportation engineers might find it necessary to determine separately the population growth trends of a city and adjacent suburb. The population of the urban area is declining with time according to

$$P_u(t) = P_{u,\max} e^{-k_u t} + P_{u,\min}$$

while the suburban population is growing, as in

$$P_s(t) = \frac{P_{s,\max}}{1 + [P_{s,\max}/P_0 - 1]e^{-k_s t}}$$

where $P_{u,\max}$, k_u, $P_{s,\max}$, P_0, and k_s = empirically derived parameters. Determine the time and corresponding values of $P_u(t)$ and $P_s(t)$ when the suburbs are 20% larger than the city. The parameter values are $P_{u,\max} = 75{,}000$, $k_u = 0.045/\text{yr}$, $P_{u,\min} = 100{,}000$ people, $P_{s,\max} = 300{,}000$ people, $P_0 = 10{,}000$ people, $k_s = 0.08/\text{yr}$. To obtain your solutions, use **(a)** graphical and **(b)** false-position methods.

[2] - Solve problem 5.22(b) in the textbook, with es = 0.05%, as follows:

Produce code to implement the modified false position method using your favorite programming language (VBA/Excel, C++, Matlab, Etc.). Your code should consists of a main program (say *MainFalsePosition*), and a function (*ModFalsePos*):

The main program should do the following:

(1) input the following values:
- *xl*, the lowest value in a bracket for the modified false position method
- *xu*, the upper value in a bracket for the modified false position method
- *es*, the percent error tolerance for convergence
- *imax,* the maximum allowed number of iterations before declaring a divergent process
- Other parameters in the problem ($P_{u,max}$, k_u, $P_{u,min}$, $P_{s,max}$, k_s, and P_0) can be defined in code, e.g., *Pumax* = 75000, etc.

(2) check if *xl* and *xu* are suitable for the modified false position method, i.e., check that *f*(*xl*)**f*(*xu*) < 0. If not, the program should report this situation as an error and stop..
(3) call subroutine or function *ModFalsePos* (described below) to return a solution
(4) outputs the solution returned by *ModFalsePos*, the number of iterations required to solve the equation, the percent relative error of the approximation at the point that the solution was found, and the value of the function *f*(*x*) at the solution point (i.e., the values *xr, iter, ea* from *ModFalsePos*, and *f*(*xr*)).

Solution:

```
Problem 2:
                        Find when Ps = 1.2 Pu
t          Ps(t)      Pu(t)      difference      loops      error
39.9873    134885     112405     -0.000517399    5          0.00139414
----------------------------------------------------------------------
```

assign2.h:

```
#pragma once
#include <functional>

namespace assign2 {
    struct Estimate {
        double error;
        double value;
        int loops;
        Estimate(double est = 0) {
            error = 100;
            loops = 0;
            value = est;
        }
    };

    Estimate ModFalsePos(std::function<double(double)>, double, double,
double, double);

    void Problem2();
}
```

assign2.cpp:
```
//----------------------------------------------------------------------------
------+
// Estimate ModFalsePos
|
// Given an arbitrary polynomial, use the modified false position method
|
// The equation "f(root) = 0", assuming l_bound < root < r_bound
|
//----------------------------------------------------------------------------
------+
Estimate assign2::ModFalsePos(
                std::function<double(double)> f,
                double l_bound,
                double u_bound,
                double max_loops,
                double max_error) {
    Estimate est(l_bound);
    double fl = f(l_bound);
    double fu = f(u_bound);
    double fr, prev, sign;
    int iu = 0, il = 0; // Horrible names, oh well

    if (fl * fu >= 0) {
        std::cout <<
            "Error: left-hand and right-hand estimates do not bound a root"
<< std::endl;
        return est;
    }

    do {
        ++est.loops;
        prev = est.value;

        est.value = u_bound - fu * (l_bound - u_bound) / (fl - fu);
        fr = f(est.value);

        if (est.value != 0) {
            est.error = abs(est.value - prev) * 100 / est.value;
        }

        sign = fl * fr;
        if (sign < 0) {
            u_bound = est.value;
            fu = f(u_bound);
            iu = 0;
            ++il;
            if (il >= 2) {
                fl /= 2;
            }
        } else if (sign > 0) {
            l_bound = est.value;
            fl = f(l_bound);
            il = 0;
            ++iu;
            if (iu >= 2) {
                fu /= 2;
```

```
            }
        } else {
            est.error = 0;
        }
    } while (est.loops < max_loops && est.error >= max_error);

    return est;
}

//-----------------------------------------------------------------------------
------+
// Problem 2 - Using the Modified False Position Method
|
//-----------------------------------------------------------------------------
------+
void assign2::Problem2() {
    const int
        Pu_max = 75000,
        Pu_min = 100000,
        Ps_max = 300000,
        P0 = 10000;
    const double
        ku = .045,
        ks = .08;

    auto Pu = [&Pu_max, &Pu_min, &ku](double t) -> double {
        return Pu_max * exp(-ku * t) + Pu_min;
    };
    auto Ps = [&Ps_max, &P0, &ks](double t) -> double {
        return Ps_max / (1 + ((Ps_max / (P0 - 1)) * exp(-ks * t)));
    };
    auto seek = [&Ps, &Pu](double t) -> double{
        return Ps(t) - 1.2 * Pu(t);
    };

    const int
        TITLE = 70,
        COL_ROOT = 10,
        COL_PVALUE = 15,
        COL_LOOPS = 10,
        COL_ERROR = 15;

    Estimate est;

    std::cout << "Problem 2:" << std::endl
        << std::setw(TITLE) << centered("Find when Ps = 1.2 Pu")
        << std::endl << std::left
        << std::setw(COL_ROOT) << "t"
        << std::setw(COL_ROOT) << "Ps(t)"
        << std::setw(COL_ROOT) << "Pu(t)"
        << std::setw(COL_PVALUE) << "difference"
        << std::setw(COL_LOOPS) << "loops"
        << std::setw(COL_ERROR) << "error"
        << std::endl;

    est = ModFalsePos(seek, 0, 100, 100, 0.05);
    std::cout
```

```
        << std::setw(COL_ROOT) << est.value
        << std::setw(COL_ROOT) << Ps(est.value)
        << std::setw(COL_ROOT) << Pu(est.value)
        << std::setw(COL_PVALUE) << seek(est.value)
        << std::setw(COL_LOOPS) << est.loops
        << std::setw(COL_ERROR) << est.error
        << std::endl;

    std::cout << "----------------------------------------------------------
-----------" << std::endl << std::endl;
}
```

Problem 3

6.30 You are designing a spherical tank (Fig. P6.30) to hold water for a small village in a developing country. The volume of liquid it can hold can be computed as

$$V = \pi h^2 \frac{[3R - h]}{3}$$

where V = volume (m^3), h = depth of water in tank (m), and R = the tank radius (m).

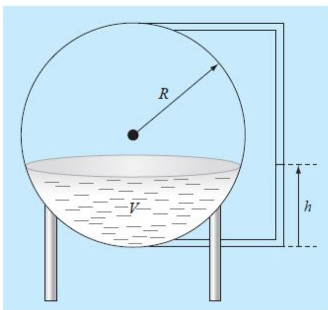


Figure P6.30

If $R = 3$ m, what depth must the tank be filled to so that it holds 30 m^3? Use three iterations of the Newton-Raphson method to determine your answer. Determine the approximate relative error after each iteration. Note that an initial guess of R will always converge.

[3] - Solve problem 6.30 in the textbook, with es = 0.05%, as follows:

Produce code to implement the Newton-Raphson method using your favorite programming language (VBA/Excel, C++, Matlab, Etc.). Your code should consists of a main program (say *MainNewtonRaphson*), and a function (*NewtonRaphson*):

The main program should do the following:

(1) input the following values:
- *x0*, initial guess for the Newton-Raphson method
- *es*, the percent error tolerance for convergence
- *imax*, the maximum allowed number of iterations before declaring a divergent process

(2) calls function *NewtonRaphson* to return a solution
(3) outputs the solution returned by *NewtonRaphson*, the number of iterations required to solve the equation, the percent relative error of the approximation at the point that the solution was found, and the value of the function *f*(*x*) at the solution point (i.e., the values *xr*, *iter*, *ea* from *NewtonRaphson*, and *f*(*xr*)).

```
Problem 3:
Root: 2.02691
Error: 9.0638e-005
Loops: 4
```

assign2.h:

```
#pragma once
#include <functional>
#include <vector>

namespace assign2 {
    Estimate FixedPoint(std::function<double(double)>, double, double,
double);

    void Problem3();
}
```

assign2.cpp:

```
//---------------------------------------------------------------------------
------+
// Estimate FixedPoint
|
// The function accepted should return x_next given the current estimate of x
|
// Iterate using the fixed point method to approximate an unknown function's
root  |
//---------------------------------------------------------------------------
------+
```

```
Estimate assign2::FixedPoint(
                std::function<double(double)> f,
                double guess,
                double max_loops,
                double max_error) {
    Estimate est(guess);
    double prev;

    do {
        ++est.loops;
        prev = est.value;

        est.value = f(est.value);
        if (est.value != 0) {
            est.error = abs(est.value - prev) * 100 / est.value;
        }
    } while (est.loops < max_loops && est.error >= max_error);

    return est;
}

//-----------------------------------------------------------------------------
------+
// Problem 3 - Using the NewtonRaphson Method
|
//-----------------------------------------------------------------------------
------+
void assign2::Problem3() {
    const double guess = 4;
    auto f = [](double h) -> double {
        return M_PI * pow(h, 2) * (9 - h) / 3 - 30;
    };
    auto fp = [](double h) -> double {
        return M_PI * (6 * h - pow(h, 2));
    };

    // The Newton-Raphson method
    auto next = [&f, &fp](double x) -> double {
        return x - (f(x) / fp(x));
    };

    Estimate root = FixedPoint(next, guess, 100, 0.05);
    std::cout
        << "Problem 3:" << std::endl
        << "Root: " << root.value << std::endl
        << "Error: " << root.error << std::endl
        << "Loops: " << root.loops << std::endl
        << std::endl
        << "--------------------------------------------------" << std::endl
<< std::endl;
}
```

Problem 4

[4] - Solve problem 6.18 in the textbook, with es = 0.05%, and δ = 0.001, as follows:

Produce code to implement the modified secant method using your favorite programming language (VBA/Excel, C++, Matlab, Etc.). Your code should consists of a main program (say MainModSecant), and a function (ModSecant):

The main program should do the following:

(1) input the following values:

- *x0*, initial guess for the modified secant method
- *δ*, increment factor for the modified secant method
- *es*, the percent error tolerance for convergence
- *imax*, the maximum allowed number of iterations before declaring a divergent process

(2) call function *ModSecant* to return a solution
(3) outputs the solution returned by *ModSecant*, the number of iterations required to solve the equation, the percent relative error of the approximation at the point that the solution was found, and the value of the function *f*(*x*) at the solution point (i.e., the values *xr*, *iter*, *ea* from *ModSecant*, and *f*(*xr*)).

6.18 A mass balance for a pollutant in a well-mixed lake can be written as

$$V\frac{dc}{dt} = W - Qc - kV\sqrt{c}$$

Given the parameter values $V = 1 \times 10^6 \text{m}^3$, $Q = 1 \times 10^5 \text{ m}^3/\text{yr}$, $W = 1 \times 10^6 \text{ g/yr}$, and $k = 0.25 \text{ m}^{0.5}/\text{g}^{0.5}/\text{yr}$, use the modified secant method to solve for the steady-state concentration. Employ an initial guess of $c = 4 \text{ g/m}^3$ and $\delta = 0.5$. Perform three iterations and determine the percent relative error after the third iteration.

```
Problem 4:
Root: 4.62408
Error: 1.4948e-005
Loops: 3
```

assign2.cpp (assign.h is the same appearance as in problem 3):

```
//----------------------------------------------------------------------------
------+
// Problem 4 - Using the Modified Secant Method
|
//----------------------------------------------------------------------------
------+
void assign2::Problem4() {
    const double
        guess = 4,
        delta = 0.001,
        V = pow(10, 6),
        Q = pow(10, 5),
        W = V,
        k = 0.25;

    auto f = [&V, &Q, &W, &k](double c) -> double {
        return (W / V) - (Q * c / V) - (k * sqrt(c));
    };

    // The modified secant method
    auto next = [&f, &delta](double x) -> double {
```

```
        return x - (delta * x * f(x) / (f(x + delta * x) - f(x)));
    };

    Estimate root = FixedPoint(next, guess, 100, 0.05);
    std::cout
        << "Problem 4:" << std::endl
        << "Root: " << root.value << std::endl
        << "Error: " << root.error << std::endl
        << "Loops: " << root.loops << std::endl
        << std::endl
        << "--------------------------------------------------" << std::endl
<< std::endl;
}
```

Problem 5

7.19 In control systems analysis, transfer functions are developed that mathematically relate the dynamics of a system's input to its output. A transfer function for a robotic positioning system is given by

$$G(s) = \frac{C(s)}{N(s)} = \frac{s^3 + 12.5s^2 + 50.5s + 66}{s^4 + 19s^3 + 122s^2 + 296s + 192}$$

where $G(s)$ = system gain, $C(s)$ = system output, $N(s)$ = system input, and s = Laplace transform complex frequency. Use a numerical technique to find the roots of the numerator and denominator and factor these into the form

$$G(s) = \frac{(s + a_1)(s + a_2)(s + a_3)}{(s + b_1)(s + b_2)(s + b_3)(s + b_4)}$$

where a_i and b_i = the roots of the numerator and denominator, respectively.

Matlab:

```
>> p = [1 12.5 50.5 66]
p =
    1.0000   12.5000   50.5000   66.0000

>> r = roots(p)
r =
   -5.5000
   -4.0000
   -3.0000

>> p = [1 19 122 296 192]
p =
     1    19   122   296   192

>> r = roots(p)
r =
   -8.0000
   -6.0000
   -4.0000
   -1.0000
```

Problem 6

[6] – Select one of the following problems (to your liking), and solve it using either *Goal Seek* or the *Solver* in *Excel, Goal Seek* in *CALC*, or using *fzero* in *Matlab*, or *fsolve* in *Scilab*.

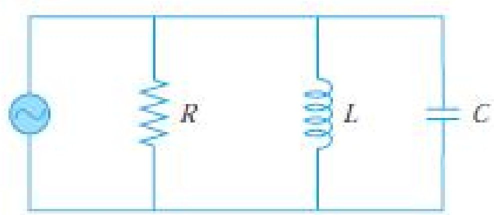
- For *Biological Engineering*, solve problem 8.7 (use pressure $p = 6500$ kPa, instead of the vlue of 65,000 kPa given in the problem statement)
- For *Civil and Environmental Engineering,* solve problem 8.17
- For *Electrical Engineering*, solve problem 8.32

8.32 Figure P8.32 shows a circuit with a resistor, an inductor, and a capacitor in parallel. Kirchhoff's rules can be used to express the impedance of the system as

$$\frac{1}{Z} = \sqrt{\frac{1}{R^2} + \left(\omega C - \frac{1}{\omega L}\right)^2}$$

where Z = impedance (Ω) and ω = the angular frequency. Find the ω that results in an impedance of 75 Ω using both bisection and false position with initial guesses of 1 and 1000 for the following parameters: $R = 225\ \Omega$, $C = 0.6 \times 10^{-6}$ F, and $L = 0.5$ H. Determine how many iterations of each technique are necessary to determine the answer to $\varepsilon_s = 0.1\%$. Use the graphical approach to explain any difficulties that arise.

Figure P8.32



Params	
R	225
C	0.0000006
L	0.5
Variable	
Omega	157.90887
Result	
Z	74.999998