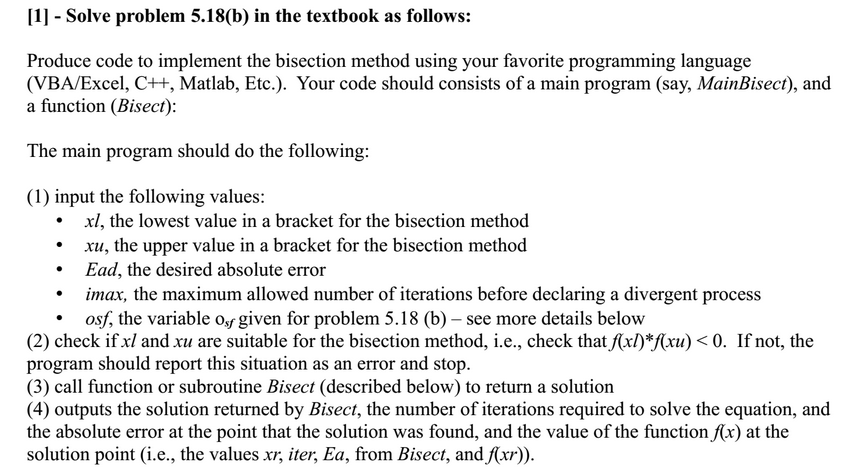
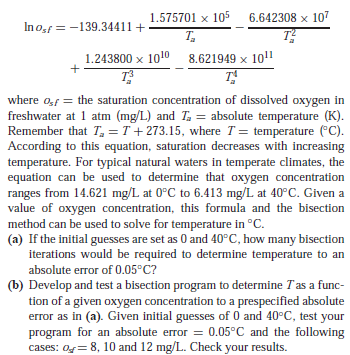
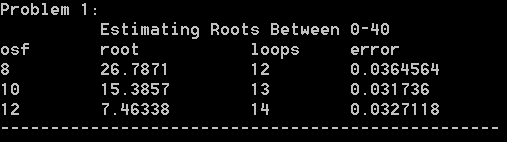
Problem 1



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;

}

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;

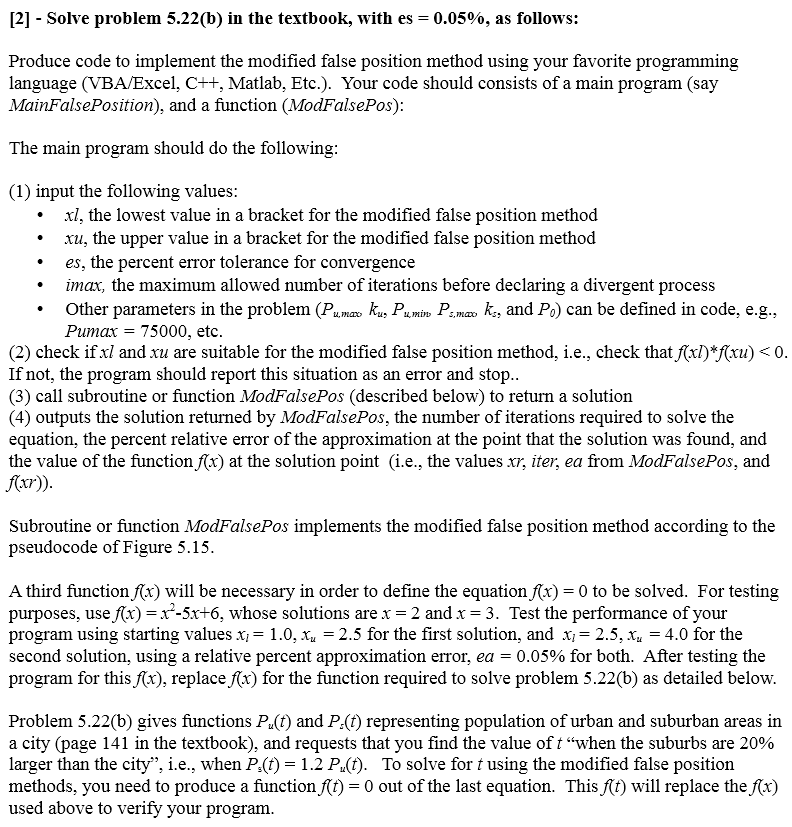
}

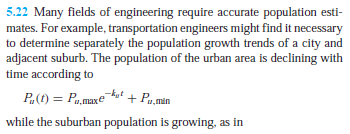
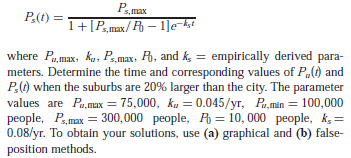
};

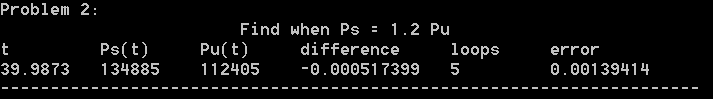
Estimate Bisect(std::function<double(double)>, double, double, double, double);

void Problem1();

}

Problem 2



Solution:

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)

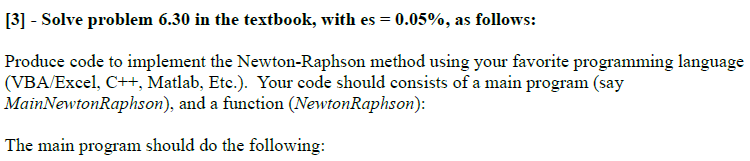
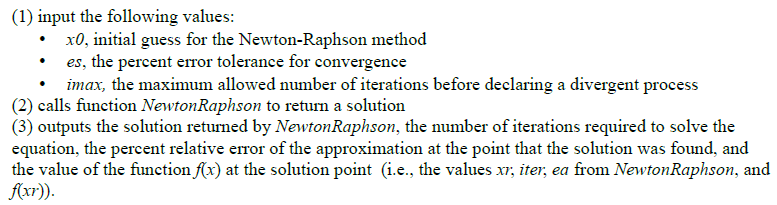
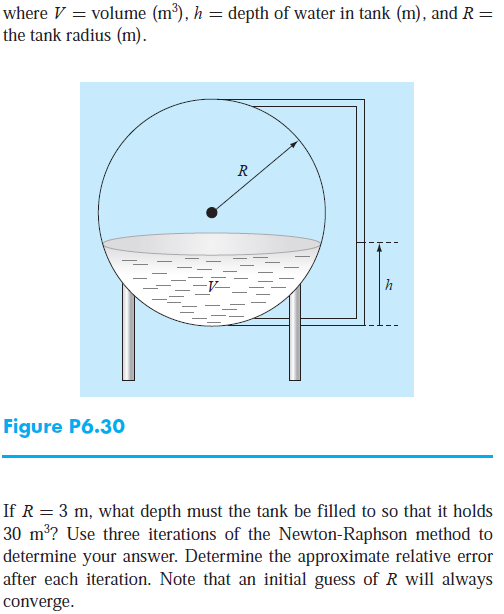
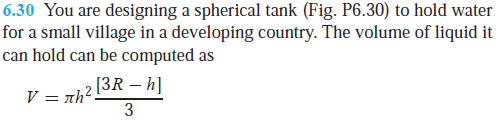
<< std::setw(COL\_LOOPS) << est.loops

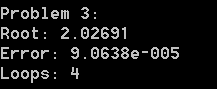
<< std::setw(COL\_ERROR) << est.error

<< std::endl;

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

}

Problem 3



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

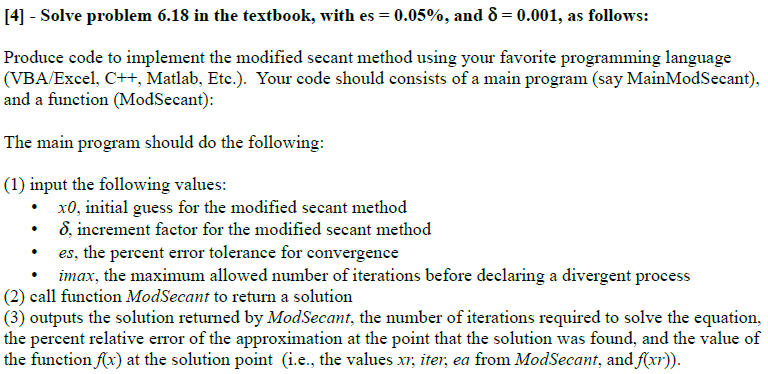
<< "Loops: " << root.loops << std::endl

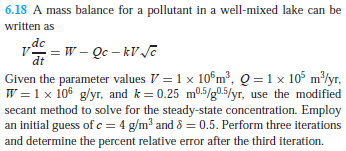
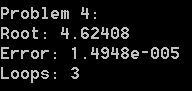
<< std::endl

<< "--------------------------------------------------" << std::endl << std::endl;

}

Problem 4





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

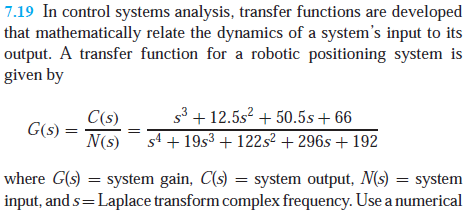
<< "Loops: " << root.loops << std::endl

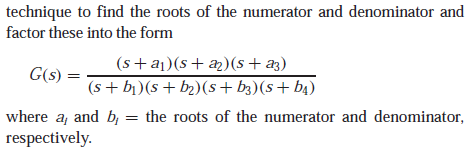
<< std::endl

<< "--------------------------------------------------" << std::endl << std::endl;

}

Problem 5





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

