Problems 1 and 2

By now, it should be clear how I handle main.cpp, assign5.hpp, and in assign5.cpp void assign5::main(). Since these problems are small and simple enough, I kept all other code in assign5.cpp for this assignment.

Common code:

double Trapm(const std::vector<double> f, double h) {

int n = f.size();

double sum = f[0];

for (int i = 1; i < n; ++i) {

sum += 2 \* f[i];

}

sum += f[n];

return h \* sum / 2;

}

double Trapm(const std::vector<double> x, const std::vector<double> y) {

int n = x.size();

double sum = 0;

for (int i = 1; i < n; ++i) {

sum += (y[i] + y[i - 1]) \* (x[i] - x[i - 1]) / 2;

}

return sum;

}

double Trapm(std::function<double(double)> f, double a, double b, int n) {

double h = (b - a) / n;

double sum = f(a);

for (int i = 1; i < n; ++i) {

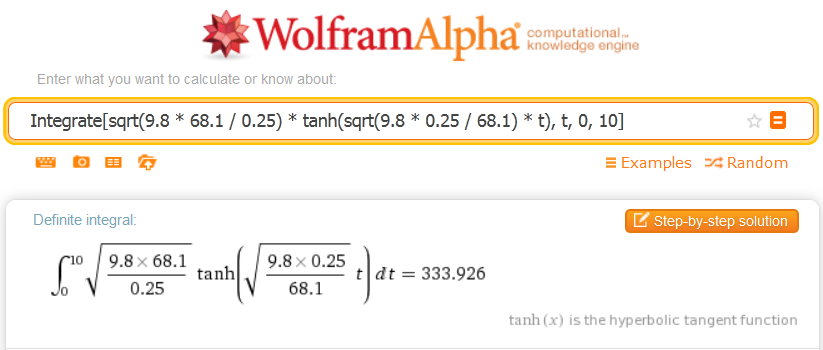
sum += 2 \* f(a + h \* i);

}

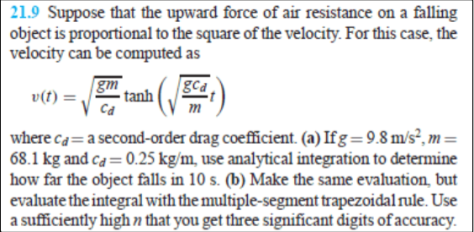
sum += f(b);

return h \* sum / 2;

}

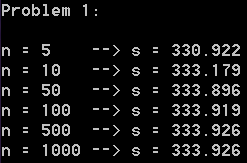


Part a: since we can use regular calculators I figured wolfram alpha would be fine.



Problem 1

Part b



Code for problem 1:

void Prob1() {

const double a = 0, b = 10,

g = 9.8, m = 68.1, c = 0.25;

auto v = [&g, &m, &c](double t) {

return sqrt(g \* m / c) \*

tanh(sqrt(g \* c / m) \* t);

};

std::cout

<< "Problem 1:" << std::endl << std::endl

<< "n = 5 --> s = " << Trapm(v, a, b, 5) << std::endl

<< "n = 10 --> s = " << Trapm(v, a, b, 10) << std::endl

<< "n = 50 --> s = " << Trapm(v, a, b, 50) << std::endl

<< "n = 100 --> s = " << Trapm(v, a, b, 100) << std::endl

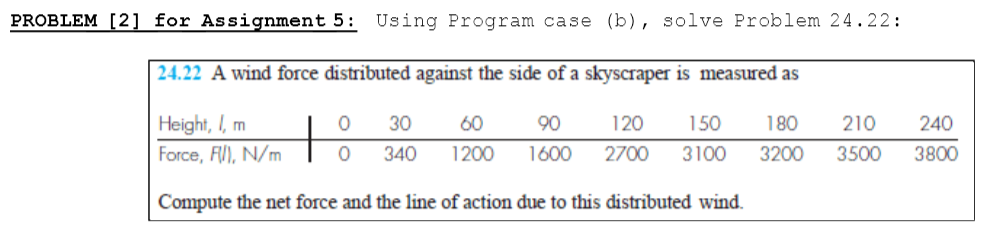
<< "n = 500 --> s = " << Trapm(v, a, b, 500) << std::endl

<< "n = 1000 --> s = " << Trapm(v, a, b, 1000) << std::endl

<< std::endl;

}

Problem 2



Code for problem 2:

void Prob2() {

const std::vector<double> x{0, 30, 60, 90, 120, 150, 180, 210, 240};

const std::vector<double> y{0, 340, 1200, 1600, 2700, 3100, 3200, 3500, 3800};

std::cout << "Problem 2: " << Trapm(x, y) << std::endl << std::endl;

}

Problems 3 and 4

Common code:

double Trap(double h, double f0, double f1) {

return h \* (f1 + f0) / 2;

}

double Simp38(double h, double f0, double f1, double f2, double f3) {

return 3 \* h \* (f0 + 3 \* f1 + 3 \* f2 + f3) / 8;

}

double Simp13m(double h, int n, const std::vector<double> y) {

double sum = y[0];

for (int i = 1; i < n - 2; i += 2) {

sum += 4 \* y[i] + 2 \* y[i + 1];

}

sum += 4 \* y[n - 1] + y[n];

return h \* sum / 3;

}

double SimpInt(const std::vector<double> y, double h) {

int n = y.size() - 1;

if (n == 1) {

return Trap(h, y[1], y[0]);

} else {

double sum = 0; int m = n;

if (n % 2 != 0) {// if we got here n > 1 is a given

sum += Simp38(h, y[n - 3], y[n - 2], y[n - 1], y[n]);

m -= 3;

}

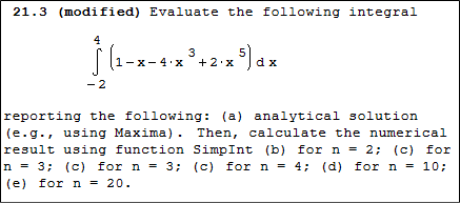
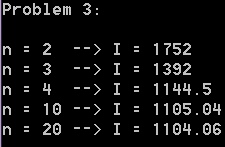
if (m > 1) { sum += Simp13m(h, m, y); }

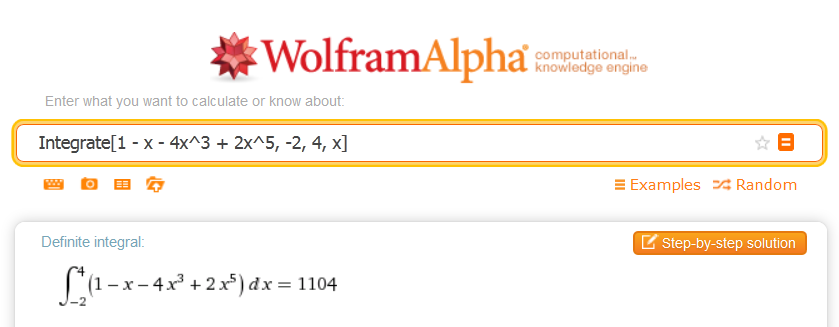
return sum;

}

}

Problem 3



Code for problem 3:

void Prob3() {

double a = -2, b = 4, h = 0;

std::vector<double> y;

auto f = [](double x) {

double xpow[6];

Powers(xpow, 6, x);

return 1 - x - 4 \* xpow[3] + 2 \* xpow[5];

};

auto make = [&a, &b, &y, &f](int n, double& h) {

y.resize(n + 1);

h = (b - a) / n;

y[0] = f(a);

for (int i = 1; i < n; ++i) {

y[i] = f(a + h \* i);

}

y[n] = f(b);

};

std::cout << "Problem 3:" << std::endl << std::endl; make(2, h);

std::cout << "n = 2 --> I = " << SimpInt(y, h) << std::endl; make(3, h);

std::cout << "n = 3 --> I = " << SimpInt(y, h) << std::endl; make(4, h);

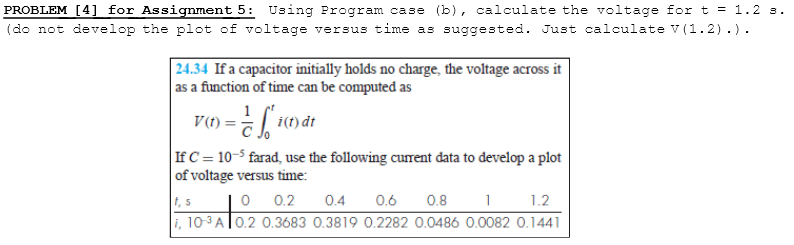
std::cout << "n = 4 --> I = " << SimpInt(y, h) << std::endl; make(10, h);

std::cout << "n = 10 --> I = " << SimpInt(y, h) << std::endl; make(20, h);

std::cout << "n = 20 --> I = " << SimpInt(y, h) << std::endl << std::endl;

}

Problem 4:



Code for problem 4:

void Prob4() {

const double h = 0.2;

const std::vector<double> y{0.2, 0.3683, 0.3819, 0.2282, 0.0486, 0.0082, 0.1441};

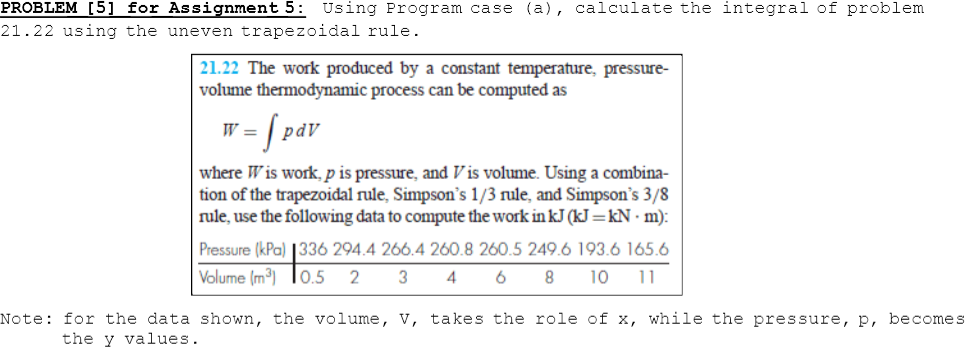
std::cout << "Problem 4: " << 100 \* SimpInt(y, h) << std::endl << std::endl;

}

Problems 5 and 6

Common code:

If you look at problems 1-2, the declaration of Trapm that accepts x and y is actually Trapun. I find it ridiculous to impose the assumption that data is equally spaced and I naturally programmed it to handle uneven data. No efficiency is lost this way, and it’s just 100 times better, cleaner code.



Code for problem 5:

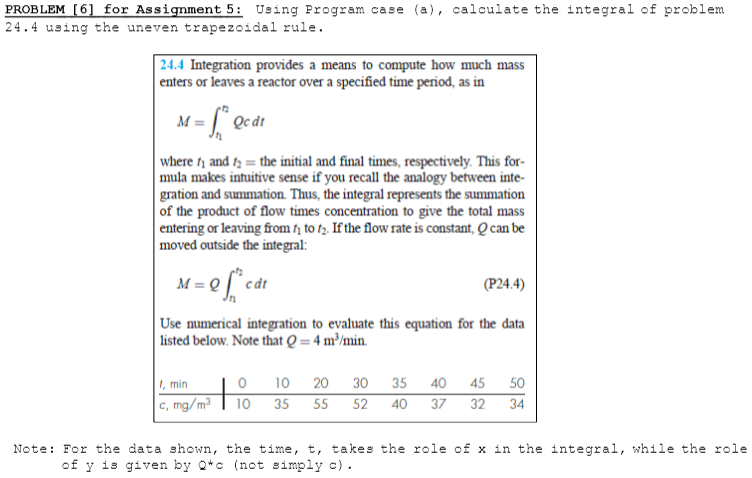
void Prob5() {

const std::vector<double> v{0.5, 2, 3, 4, 6, 8, 10, 11};

const std::vector<double> p{336, 294.4, 266.4, 260.8, 260.5, 249.6, 193.6, 165.6};

std::cout << "Problem 5: " << Trapm(v, p) << std::endl << std::endl;

}



Code for problem 6

void Prob6() {

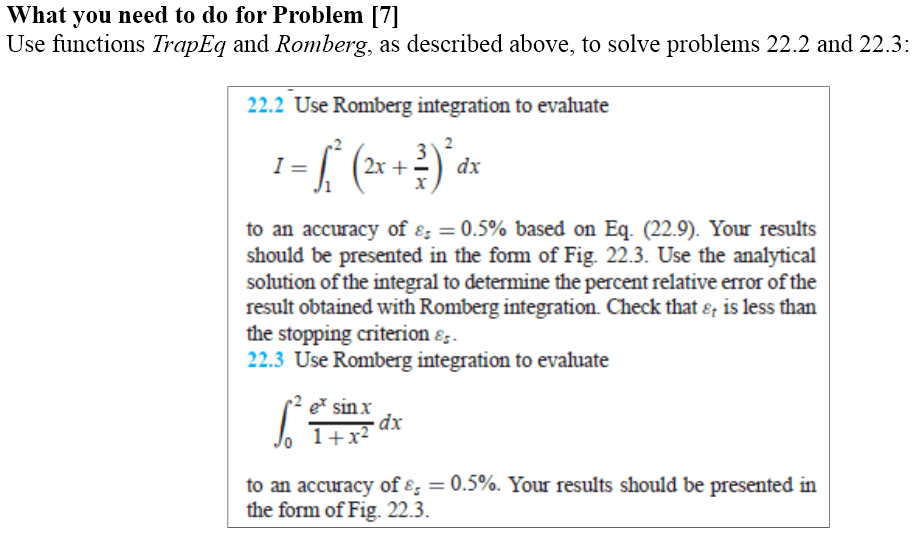
const std::vector<double> t{0, 10, 20, 30, 35, 40, 45, 50};

const std::vector<double> c{10, 35, 55, 52, 40, 37, 32, 34};

std::cout << "Problem 6: " << 4 \* Trapm(t, c) << std::endl << std::endl;

}

Problem 7



Output:

ans =

1.

ea1 =

0.0097823

iter1 =

2.

n1 =

4.

II =

25.834565

ea2 =

0.0997471

iter2 =

2.

n2 =

4.

I2 =

1.941836

RombergScript.sce

diary("Problems22\_22\_And\_22\_3.txt")

//Problem 22.2

a = 1; b = 2; maxiter = 50; ea = .5;

function [y] = f2(x)

y = (2\*x + (3 / x))^2;

endfunction;

[II, n1, iter1, ea1] = Romberg(a,b,maxiter,ea,f2)

//Problem 22.3

a = 0; b = 2; maxiter = 50; ea = .5;

function [z] = f3(t)

z = (%e^(t) \* sin(t)) / (1 + t^2);

endfunction;

[I2,n2,iter2,ea2] = Romberg(a,b,maxiter,ea,f3)

diary(0)