REPORT OF PROJECT 3

The following document contains:

• Comments and explanations that I think are necessary for understanding my program.

• The output of my program according to the tasks.

• Program listing.

Alejandra Velasco Sánchez-Villares

Explanations & Listing

Grid.h

#ifndef GRID\_H

#define GRID\_H

#include<stdlib.h>

#include<cmath>

class Curvebase {

public:

// Constructor

Curvebase(const double pmin = 0.0, const double pmax = 1.0);

// Arc Length Parametrizations

virtual const double x(const double s);

virtual const double y(const double s);

protected:

// Minimum and Maximum Values for ‘p’.

const double pmin\_, pmax\_;

// Orientation of the curve

bool rev;

// Arc Length

double length;

// Pure Virtual Functions

virtual const double xp(const double p) const = 0;

virtual const double yp(const double p) const = 0;

virtual const double dxp(const double p) const = 0;

virtual const double dyp(const double p) const = 0;

// Newton Method

const double newton (const double a, const double toln, const double s);

// Integration Method from Project 1

typedef const double (Curvebase ::\*FunctionPointer)(const double&)const; //define type of "FunctionPointer" (const double)(Curvebase::\*)(const double) const

// Adaptive Simpson Integration Function

const double ASI (const FunctionPointer f, const double& pmin, const double& p, const double& toli) const;

// Integrated Function

const double func (const double& x) const;

// Simpson Rule

const double I (const FunctionPointer f, const double& a, const double& b) const;

};

// Straight Lines of the figure we are given.

class Lines : public Curvebase {

public:

// Constructor

Lines(const double pmin, const double pmax, const double val, const bool dir);

// Re-used Functions

const double x(const double s);

const double y(const double s);

private:

// 0 for y=val\_, 1 for x=val\_

const bool dir\_;

const double val\_;

// Re-used Functions

const double xp(const double p)const;

const double yp(const double p)const;

const double dxp(const double p)const;

const double dyp(const double p)const;

};

// Bottom Curve of the figure we are given.

class Bcurve : public Curvebase {

public:

// Constructor

Bcurve(const double pmin, const double pmax) : Curvebase(pmin, pmax){

if (rev) std::cout << "input error at construction of Bcurve" << std::endl;

}

private:

// Re-used Functions

const double xp(const double p)const;

const double yp(const double p)const;

const double dxp(const double p)const;

const double dyp(const double p)const;

};

// Domain Class

class Domain {

public:

// Constructor

Domain(Curvebase& s0, Curvebase& s1, Curvebase& s2, Curvebase& s3);

void generate\_grid(const int m, const int n);

void write\_grid()const;

~Domain();

protected:

// Static bind

Curvebase \*sides[4];

int m\_ = 0, n\_ = 0;

// Grid Data

double \*x\_, \*y\_;

const double phi1(const double x) const { return 1-x; }

const double phi2(const double x) const { return x; }

// Consistency Check

bool check\_consistency()const;

};

#endif

Grid.cpp

#include<iostream>

#include<cmath>

#include<stdlib.h>

#include<cstdio>

#include "Grid.h"

// ‘toln’ is 0.01\*(h1,h2) for Newton Function and ‘toli’ for Integral Function

double toln = 0.01 / 50.

toli = 0.1\*toln;

// Constructor of Curvebase

Curvebase::Curvebase(const double pmin, const double pmax) : pmin\_(pmin), pmax\_(pmax)

{

if (pmin > pmax) {

rev = true;

}

Else {

rev = false;

}

}

// Linear Initial Guess

const double Curvebase::x(const double s) {

if (s < 0 || s > 1) exit(EXIT\_FAILURE);

double p = newton((1 - s)\*pmin\_ + s \* pmax\_, toln, s);

return xp(p);

}

const double Curvebase::y(const double s) {

if (s < 0 || s > 1) exit(EXIT\_FAILURE);

double p = newton((1 - s)\*pmin\_ + s \* pmax\_, toln, s);

return yp(p);

}

// Newton Function

const double Curvebase::newton(const double p0\_, const double toln, const double s) {

double p0, p1;

// Arc Length

length = ASI(&Curvebase::func, pmin\_, pmax\_, toli);

p1 = p0\_;

p0 = p1 + toln + 1.;

while (abs(p1 - p0) > toln) {

p0 = p1;

//p(i+1)=p(i)-F(p(i))/F'(p(i)); F(p)=l(p)-s\*length

p1 = p0 - (ASI(&Curvebase::func, pmin\_, p0, toli) - s \* length) / sqrt(dxp(p0)\*dxp(p0) + dyp(p0)\*dyp(p0));

};

if (p1 < pmin\_ || p1 > pmax\_) exit(EXIT\_FAILURE);

return p1;

}

const double Curvebase::ASI(const FunctionPointer f, const double& a, const double& b, const double& tol) const{

double I1, I2, errest;

I1 = I(f, a, b);

// I(alpha,gamma) + I(gamma,beta)

I2 = I(f, a, (a + b) / 2.) + I(f, (a + b) / 2., b);

errest = abs(I1 - I2);

if (errest < 15.\*tol) return I2;

// Recursion

return ASI(f, a, (a + b) / 2., tol / 2.) + ASI(f, (a + b) / 2., b, tol / 2.);

}

const double Curvebase::func(const double& x) const {

return sqrt(dxp(x)\*dxp(x) + dyp(x)\*dyp(x)); //

}

const double Curvebase::I(const FunctionPointer f, const double& a, const double& b) const{

return (b - a) / 6.\*((this->\*f)(a) + 4.\*(this->\*f)((a + b) / 2.) + (this->\*f)(b));

}

// Lines Constructor

Lines::Lines(const double pmin, const double pmax, const double val, const bool dir) : Curvebase(pmin, pmax), val\_(val), dir\_(dir) {

length = abs(pmin - pmax);

}

const double Lines::x(const double s){ //overwrite base

if (s < 0 || s > 1) exit(EXIT\_FAILURE);

if (!dir\_) {

if (!rev) {//normal orientatinon

return xp(pmin\_ + s \* length);

}

else {//reverse orientatinon

return xp(pmin\_ - (1 - s) \* length);

}

}

else {

return val\_;

}

}

const double Lines::y(const double s) { //overwrite base

if (s < 0 || s > 1) exit(EXIT\_FAILURE);

if (dir\_) {

if (!rev) {//normal orientatinon

return yp(pmin\_ + s \* length);

}

else {//reverse orientatinon

return yp(pmin\_ - (1 - s) \* length);

}

}

else {

return val\_;

}

}

const double Lines::xp(const double p)const {

if (!dir\_) {

return p;

}

else {

return val\_;

}

}

const double Lines::yp(const double p)const {

if (!dir\_) {

return val\_;

}

else {

return p;

}

}

const double Lines::dxp(const double p)const {

if (!dir\_) {

return 1.;

}

else{

return 0.;

}

}

const double Lines::dyp(const double p)const {

if (!dir\_) {

return 0.;

}

else{

return 1.;

}

}

// Bcurve Constructors

const double Bcurve::xp(const double p) const{

return p;

}

const double Bcurve::yp(const double p) const {

if (p < -3) {

return 0.5 / (1. + std::exp(-3.\*(p + 6.)));

}

else {

return 0.5 / (1. + std::exp(3.\*p));

}

}

const double Bcurve::dxp(const double p) const {

return 1.;

}

const double Bcurve::dyp(const double p) const {

if (p < -3) {

return 1.5\*std::exp(-3.\*(p + 6.)) / (1. + std::exp(-3.\*(p + 6.))) / (1. + std::exp(-3.\*(p + 6.)));

}

else {

return -1.5 \*std::exp(3.\*p) / (1. + std::exp(3.\*p)) / (1. + std::exp(3.\*p));

}

}

// Domain

Domain::Domain(Curvebase& s0, Curvebase& s1, Curvebase& s2, Curvebase& s3) {

sides[0] = &s0; //dynamic type changed

sides[1] = &s1;

sides[2] = &s2;

sides[3] = &s3;

if (!check\_consistency())

// sides[0] = sides[1] = sides[2] = sides[3] = nullptr;

m\_ = n\_ = 0; //no grid

// x\_ = y\_ = nullptr;

}

void Domain::generate\_grid(const int m, const int n) {

if (m < 1 || n < 1) exit(EXIT\_FAILURE);

if (m\_ > 0) { // already exist grid

delete[] x\_;

delete[] y\_;

}

m\_ = m; n\_ = n;

x\_ = new double[(m\_ + 1)\*(n\_ + 1)];

y\_ = new double[(m\_ + 1)\*(n\_ + 1)];

double h1 = 1. / m, h2 = 1. / n;

for (int i = 0; i <= m\_; i++) {

for (int j = 0; j <= n\_; j++) {

int ind = i + j \* (m\_ + 1); //1D array

x\_[ind] = phi1(i\*h1)\*sides[3]->x(j\*h2) + phi2(i\*h1)\*sides[1]->x(j\*h2)

+ phi1(j\*h2)\*sides[0]->x(i\*h1) + phi2(j\*h2)\*sides[2]->x(i\*h1)

- phi1(i\*h1)\*phi1(j\*h2)\*sides[0]->x(0)

- phi2(i\*h1)\*phi1(j\*h2)\*sides[1]->x(0)

- phi1(i\*h1)\*phi2(j\*h2)\*sides[3]->x(1)

- phi2(i\*h1)\*phi2(j\*h2)\*sides[2]->x(1);

y\_[ind] = phi1(i\*h1)\*sides[3]->y(j\*h2) + phi2(i\*h1)\*sides[1]->y(j\*h2)

+ phi1(j\*h2)\*sides[0]->y(i\*h1) + phi2(j\*h2)\*sides[2]->y(i\*h1)

- phi1(i\*h1)\*phi1(j\*h2)\*sides[0]->y(0)

- phi2(i\*h1)\*phi1(j\*h2)\*sides[1]->y(0)

- phi1(i\*h1)\*phi2(j\*h2)\*sides[3]->y(1)

- phi2(i\*h1)\*phi2(j\*h2)\*sides[2]->y(1);

}

}

}

void Domain::write\_grid()const {

FILE \*fp;

fopen("grid.bin", "wb");

fwrite(x\_, sizeof(double), (m\_ + 1)\*(n\_ + 1), fp);

fwrite(y\_, sizeof(double), (m\_ + 1)\*(n\_ + 1), fp);

fclose(fp);

}

// Destructor

Domain::~Domain() {

if (m\_ > 0) {

delete[] x\_;

delete[] y\_;

}

}

bool Domain::check\_consistency() const {

double tol = 1.e-5; //permit small non-consistensy

if (abs(sides[0]->x(1) - sides[1]->x(0)) < tol && abs(sides[0]->y(1) - sides[1]->y(0)) < tol &&

abs(sides[1]->x(1) - sides[2]->x(1)) < tol && abs(sides[1]->y(1) - sides[2]->y(1)) < tol &&

abs(sides[2]->x(0) - sides[3]->x(1)) < tol && abs(sides[2]->y(0) - sides[3]->y(1)) < tol &&

abs(sides[3]->x(0) - sides[0]->x(0)) < tol && abs(sides[3]->y(0) - sides[0]->y(0)) < tol)

{

return true;

}

else {

std::cout << "check\_consistency false" << std::endl;

return false;

}

}

Main.cpp

#include<iostream>

#include "Grid.h"

#include "Grid.cpp"

int main() {

const int m = 50, n = 20;

// Lines s1(0, 3, 5, 1), s2(-10, 5, 3, 0), s3(0, 3, -10, 1);

Lines s1(3, 0, 5, 1), s2(5, -10, 3, 0), s3(0, 3, -10, 1); // pmin, pmax, val, dir

Bcurve s0(-10, 5); // pmin, pmax

Domain A(s0, s1, s2, s3);

A.generate\_grid(m, n);

std::cout << "Grid generated" << std::endl;

A.write\_grid();

std::cout << "Grid written" << std::endl;

return 0;

}

Output

