PCPP18 - Project 3

Alma Andersson

November 29, 2018

1 C++ Version and Source Code access

All code provided for this project is written as to be compatible with C++11. It has been tested and compiled with the g++ GNU compiler on an Ubuntu 16.04 LTS system.

All code is made available at github where also instructions of how to best run the program is found, presented in the README.

2 Outline

This report is divided into three parts each describing the core elements of this project. The first part outlines boundary curves later used to compile a four sided domain are constructed. The second part describes how these boundary curves are assembled into a larger the aforementioned domain. The third part finally describes how a grid is generated over the domain using *transfinite interpolation*. Only the main features of the program and classes/functions are outlined here, for more explicit information the reader is referred to the Appendix where the source code can be found.

3 Part 1 - Boundary Curve Formation

The Curvebase class

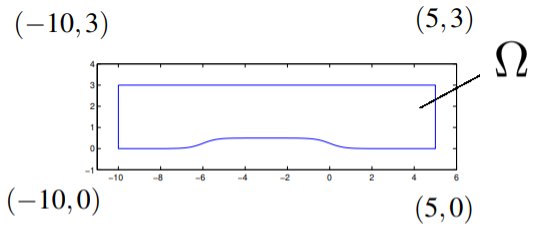
All boundary curves inherit their structure from the abstract base class *Curvebase*. This class has among it's non-virtual members the functions necessary to compute the arc-length of a curve, using adaptive simpson integration (performed using the 4 functions *integrate, function, I1 and I2*). It also contains the *solve* member function using Newton Rhapson's method to approximate the root of the expression given in equation 1,

$$s - \frac{1}{L}\int_a^{p_k} \sqrt{x'(p)^2 + y'(p)^2}dp = 0, \qquad L = \int_a^b \sqrt{x'(p)^2 + y'(p)^2}dp \tag{1}$$

Where p is the variable used to parametrize the boundary curve. Allowing the user to enter a "relative" position $s \in [0, 1]$ representing the fraction of the arc-length by which the point she wants to access is situated. As to illustrate, if $s = 0.5$ the return-value of the solve function will be the value p such that the arc-length from the lower limit to p is half of the total arc-length of the curve, note how $p \in \mathbb{R}$.

Given the parametrization $(x, y) = (x(p), y((p))$, functions to retrieve the x respectively y-coordinate, as well as the derivatives $x'(p)$ and $y'(p)$ given p, are added as virtual functions. This, since the curve parametrization is a defining property of each individual curve, and a static parametrization would be of no use in an abstract base class.

Included in Curvebase class are also a default constructur and destructor, as well as a copy constructor, allowing for new Curvebase objects to be initialized with an already existing Curvebase object. The Curvebase also includes a constructor, allowing for an object to be initialized with lower and upper end, points to be specified as well as curve orientation and tolerance level to be used in the numerical approximations (default is 10^{-5}). Finally the



The lower boundary is given by the function

$$f(x) = \begin{cases} \frac{1}{2}\frac{1}{1+exp(-3(x+6))}, & x \in [-10,-3) \\ \frac{1}{2}\frac{1}{1+exp(3x)}, & x \in [-3,5] \end{cases}$$

Figure 1: Domain Ω as defined in the assignment. (Modified picture, taken from instructions)

Curvebase class has a *partially virtual* function *setLength* which will compute the arc-length of a parametrized curve using Newton Rhapson's method and the equation described in eq 1; albeit a valid approach for all curves, simpler approaches for length calculations are available in certain special cases such as straight line segments allowing for unecessary computations to be avoided.

Horzline and Vertline classes

Two additional abstract base classes which inherits from the Curvebase are used, representing two similar and common scenarios upon domain construction, namely where the either the y-variable is a function of the x-variable and the reverse. Allowing for parametrizations on the for (i) $(x, y) = (p, f(p))$ respectively (ii) $(x, y) = (g(p), p)$. The former scenario (i) is captured by the class *Horzline* whilst *Vertline* models the latter (ii).

Both these classes thus defines the inherited virtual functions (from Curvebase) for the x and y-coordinate parametrization and the respective derivatives. Two new virtual functions are included in each class allowing for definition of the relationship between x and y-coordinates (and the derivative) to be defined in derived classes; these classes being the *afunc(p)* and *afuncd(p)* with $a = \{x, y\}$ having the former define the coordinate mapping and the latter it's derivative.

LeftRightBorder, TopBorder and BottomBorder classes

In the main-program, three classes (LeftRightBorder, TopBorder and BottomBorder) are defined, where LeftRightBorder is a derived class of *Vertline*, handling the vertical borders that are vertical straight line segments. TopBorder as well as BottomBorder are derived classes of Horzline. TopBorder handles a scenario where we have a horizontal straight line segment, whilst BottomBorder is used for cases where the y-coordinate is a function of the x-coordinate.

The LeftRightBorder class has a member function *setXpos* that sets the static x-coordinate, whilst the member function *setYpos* in TopBorder does the equivalent but for the y-coordinate. Both the LeftRightBorder and TopBorder redefines the *getLength*-function, as to simply be the euclidian distance between upper and lower endpoint.

Boundary Curves

The following parametrizations are used for the four boundary curves in the given domain Ω (see Fig 1 for reference).

- Left Border : $(x, y) = (-10, y)$
- Bottom Border : $(x, y) = (x, f(x))$, with $f(x)$ defined in Fig 1
- Right Border : $(x, y) = (5, y)$
- Top Border : $(x, y) = (x, 3)$

4 Part 2 - Domain Formation

A class named *Domain* is used to create the four sided domain. Given that the proper orientation of each boundary curve is set, the user is free to provide the boundary curves in any random order upon initialization of a Domain object. The function, *check_conistency* will automatically assign the boundary curves to the correct side. The directionality of a boundary curve is given by treating Ω as a positively oriented domain, and assigning those boundary curves where the flow from startpoint to endpoint is consistent with this orientation as positively oriented whilst those where the oppoiste is true are seen as negatively oriented.

Functions to access the orientation is included in the Curvebase base class (*ori*), also a function to return a given side of a Domain object (*getSide)* is used, as to access the boundary curves of a temporary Domain object upon sorting the domain boundaries.

5 Part 3 - Grid Generation

Once a Domain object have been initialized, the function *make_grid* allows the user to generate a structured grid with a specified number of *intervals* along the horizontal respectively vertical direction. This is done by *transfinite interpolation*, which essentially first generates a grid on the unit square $\Gamma = [0,1] \times [0,1]$ and where each point is mapped to the actual domain Ω by using linear interpolation between points known on the boundaries with either x and y-value equal to that of the point to be interpolated, as illustrated in the equation below.

$$\begin{aligned} x(s,t) = (1-s)x(0,t) + sx(1,t) + (1-t)x(s,0) + tx(s,1) \\ -s(1-t)x(1,0) - (1-s)tx(0,1) - tsx(1,1) - (1-s)(1-t)x(0,0) \end{aligned}$$

$$\begin{aligned} y(s,t) = (1-s)y(0,t) + sy(1,t) + (1-t)y(s,0) + tx(s,1) \\ -s(1-t)y(1,0) - (1-s)ty(0,1) - tsy(1,1) - (1-s)(1-t)y(0,0) \end{aligned}$$

If a grid is already present, this is erased before the rendering of a new grid. The mapping above is easily done by realizing the following

1. $x(0,t)$ and $y(0,t)$ - represents points on the left boundary curve
2. $x(1,t)$ and $y(1,t)$ - represents points on the right boundary curve
3. $x(s,0)$ and $y(s,0)$ - represents points on the bottom boundary curve
4. $x(s,1)$ and $y(s,1)$ - represents points on the top boundary curve

Given the parametrization and initialization of the boundary curve classes, all these expressions can thus easily be evaluated. The negative terms representing the corners, which of course, values also are known for (given how these are the endpoints of the boundary curves. The functions to perform the mapping are named *xmap* and *ymap* respectively, and are members of the domain class.

Additional to the above mentioned member functions, the Domain has a copy constructor and assignment operator overloading, as to allow for initialization of a Domain from another Domain object. This will copy all properties, including the grid (if such exists) to the new object.

As recommended in the instructions, *fwrite* in combination with *fopen* and *fclose* is used to save the generated grid as two bin-files, one for the x-coordinates and one for the y-coordinates. This is done by calling the function *saveCoordinates* a member of the Domain class.

A boolean value can be passed as a parameter to the *saveCoordinates* function, which if true will allow the user to specify the "stem" of the filenames used to save the coordinates should be saved, this will be appended to the prefixes "x_vec_" respectively "y_vec_". As the program is currently setup, with all source files placed in a folder "bin" the output is placed in a directory "res" positioned in the same directory as "bin". If set to false (as is

default) the names *x_vec_generated_grid.bin* and *y_vec_generated_grid.bin* are used instead.

The user can also specify that the lower boundary should be higher resolved, meaning a higher density of spots as compared to the upper boundary by using the "stretching" function given in equation 2

$$T(\sigma) = 1 + \frac{\tanh(3(1-\sigma))}{\tanh(3)} \tag{2}$$

This is done by calling the function *doLowerResolve* passing "true" as an argument, if "false" is passed no adjustment of the resolution is done (this is default).

6 Visualization

In order to visualize the results, i.e. plotting the grid a simple python-script is provided, named "visualize.py", found at same level as the *res* and *bin* folders. This script takes as first argument the path to the x-coordinate file and second argument that of the y-coordinate file. This script utilize the numpy and matplotlib python libraries (which are usually included as standard packages in most python versions).

7 Results and Comments

The program *main.cpp* is constructed as a "proof of concept" , here information about all four boundary curves is printed and the functionality of the x and y member functions is demonstrated. A domain from the four boundary curves representing those in Ω is formed and two 20×50 grids are generated, one with increased lower boundary resolution and one with no adjustment to the resolution. The user is requested to enter the names of the files to which these coordinates should be saved (.bin should be included as the extension).

Figure 2 illustrates the non-adjusted (normal) grid that was generated, visualized using the *visualize.py* script, whilst figure 3 displays the same output put for the adjusted resolution grid.

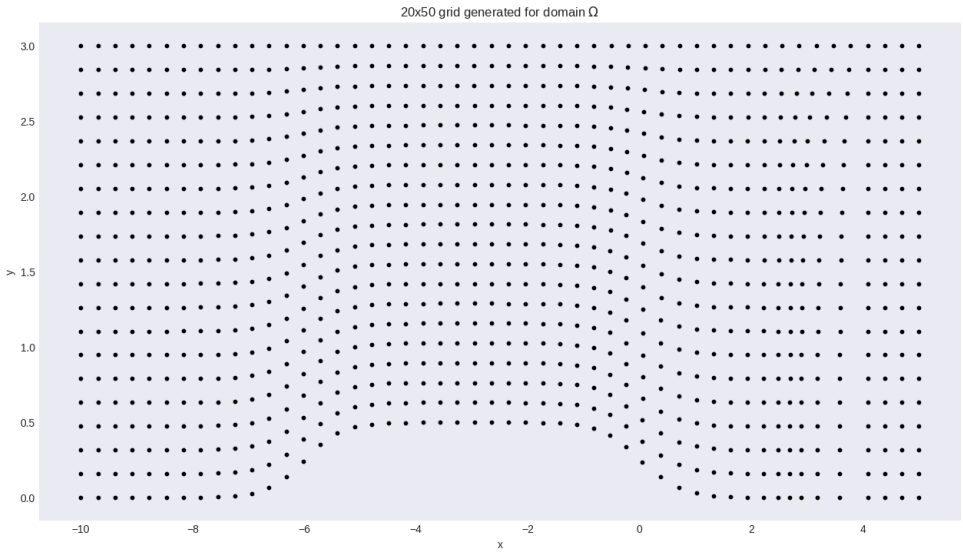


Figure 2: "Normal" grid, meaning no adjustment to the resolution is applied.

There are some artifacts to the generated grid, looking at the right side, it's possible to discern how some irregularities in the grid pattern arise. Whether this is an inherent property of the method, or a design flaw in

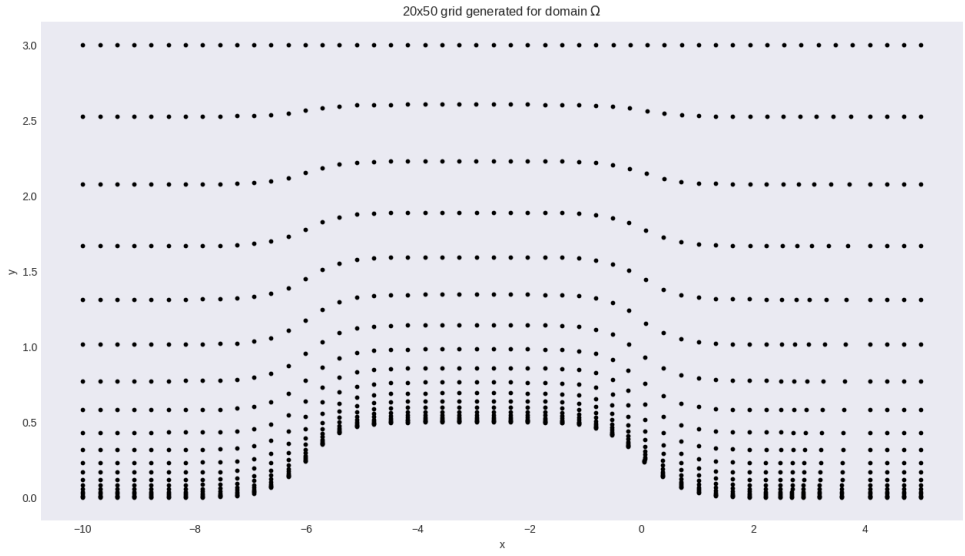


Figure 3: Adjusted resolution, with higher resolution for the lower boundary generated using by distributing the vertices in the unit-square along the vertical direction according to equation 2

the code was not clear. However given how the "bump" at the lower boundary slightly distorts the grid, it is of course expected that some compression and non-uniformity is seen, however the non-symmetrical effect (i.e. it is not observed on the left hand side) indicate that this might be due to a bug in the code.

The full output from main.cpp, as obtained when generating the two grids displayed above can be found in the Appendix (section "Output from main.cpp").

8 Appendix

8.1 Main Program - main.cpp

```
#include <iostream>
#include <cmath>
#include <cstdio>
#include <string>
#include "hline.h"
#include "vline.h"
#include "domain.h"


//class for left an right borders of Omega domain
//specific case where the boundary curve
//is a vertical straigt line segment
class LeftRightBorder : public Vertline {
        private:
                double x_pos; //fixed x-position
                double xfunc(double p); //parametrization of x-coordinate
                double xfuncd(double p); //derivative of x-coordinate parametrixation


        public:
                LeftRightBorder(double a, double b, bool dir) : Vertline(a,b, dir) {}; //use same constructor as Vertline
                ~LeftRightBorder(); //default destructor
                void setXpos(double p); //set fixed x-position
                void setLength(void); //compute total curve length
};

LeftRightBorder::~LeftRightBorder(){};

//sets fixed x-position
void LeftRightBorder::setXpos(double p) {
        x_pos = p;
        return;
};

//redefines function to compute length
void LeftRightBorder::setLength(void){
        length =  (abs(pmax-pmin));
};
//x-coordinate parametrization
double LeftRightBorder::xfunc(double p){
        return x_pos;
};
//derivative of x-coordinate parametrization
double LeftRightBorder::xfuncd(double p){
        return 0;
};

//class for top boundary curve
//specific case where boundary curve
//is a horizontal straight line segment
class TopBorder : public Horzline {
        private:
                double y_pos; //fixed y-position
                double yfunc(double p); //y-coordinate parametrization
                double yfuncd(double p);//derivative of y-coordinate parametrization
```

```
        public:
                TopBorder(double a, double b, bool dir) : Horzline(a,b,dir) {}; //use same constructor as Horzline
                ~TopBorder(); //default destructor
                void setYpos(double p); //set fixed y-position
                void setLength(void); //compute length of boundary curve
};

//default destructor
TopBorder::~TopBorder(){};


//sets fixed y-position
void TopBorder::setYpos(double p){
        y_pos = p;
        return;
};

//redefines function to compute length
void TopBorder::setLength(void){
        length = abs(pmax-pmin);
};

//y-coordinate parametrization
double TopBorder::yfunc(double p){
        return y_pos;
};

//derivative of y-coordinate parametrization
double TopBorder::yfuncd(double p){
        return 0;
};

//class for botto boundary curve
class BottomBorder : public Horzline{
        private:
                double yfunc(double p); //y-coordinate parametrization
                double yfuncd(double p);//derivative of y-coordinate parametrization
        public:
                BottomBorder(double a, double b, bool dir) : Horzline(a,b,dir) {}; //use same constructor as Horzline
                ~BottomBorder(); //default destructor
};

//default destructor
BottomBorder::~BottomBorder(){};

//y-coordinate parametrization
double BottomBorder::yfunc(double p){
        if (p < -3.0) {
                return 0.5*(1.0/(1.0 +exp(-3.0*(p+6))));

        } else if ( p >= -3.0){
                return 0.5*(1.0/(1.0 + exp(3*p)));

        } else {

                std::cout << "Bottom func: Error fetching point -> " << p << std::endl;
                return 0;
        }

};
```

```
//derivative of  y-function parametrization
double BottomBorder::yfuncd(double p){
        if (p < -3) {

                double nom = exp(-3.0*p - 18.0*p);
                double den = pow(exp(1.0 + exp(-3.0*p - 18.0*p)),2);

                return 1.5 * nom / den;

        } else if ( p >= -3){

                double nom = exp(3.0*p);
                double den = pow((1.0 + exp(3*p)),2);

                return -1.5 * nom / den;

        } else {

                std::cout << "Bottom funcd: Error fetching point -> " << p << std::endl;
                return 0;
        }

};

//main program
int main(){

        std::cout << "Task 1-2" << std::endl;
        std::cout << "\n" << "Test of Curvebase class" << std::endl;

        std::cout << "Left Border" << std::endl;
        LeftRightBorder leftb(0.0,3.0,false);
        leftb.setXpos(-10.0);
        leftb.setLength();
        leftb.printInfo();


        std::cout << "\n";
        std::cout << "Right border" << std::endl;
        LeftRightBorder rightb(0.0,3.0, true );
        rightb.setXpos(5.0);
        rightb.setLength();
        rightb.printInfo();

        std::cout << "\n" ;
        std::cout << "Bottom Border " << std::endl;
        BottomBorder botb(-10.0, 5.0,true);
        botb.setLength();
        botb.printInfo();


        std::cout << "\n";
        std::cout << "Top Border" << std::endl;
        TopBorder topb(-10.0, 5.0, false);
        topb.setYpos(3.0);
        topb.setLength();
        topb.printInfo();

        std::cout << "\n" << std::endl;
        std::cout << "Task 3-5" << std::endl;
        std::cout << "using above defined boundary curves to generate domain Omega" << std::endl;
```

```
    Domain omega(botb, topb, rightb, leftb);

    int n_rows = 50;
    int n_cols = 20;

    omega.make_grid(n_rows, n_cols);
    omega.saveCoordinates(true);
    std::cout << "saved x and y coordinates of normal grid" << std::endl;

    omega.doLowerResolve(true);
    omega.make_grid(n_rows, n_cols);
    omega.saveCoordinates(true);
    std::cout << "saved x and y coordinates of lower boundary resolved grid" << std::endl;


    return 0;
}
```

Output from main_p1.out (Part 1)

```
Task 1-2

Test of Curvebase class
Left Border
lower boundary : > 0
upper boundary : > 3
total length : > 3
orientation : > 0
x(0.5) : > -10
y(0.5) : > 2

Right border
lower boundary : > 0
upper boundary : > 3
total length : > 3
orientation : > 1
x(0.5) : > 5
y(0.5) : > 2

Bottom Border
lower boundary : > -10
upper boundary : > 5
total length : > 15
orientation : > 1
x(0.5) : > -1.84082
y(0.5) : > 0.49801

Top Border
lower boundary : > -10
upper boundary : > 5
total length : > 15
orientation : > 0
x(0.5) : > -2
y(0.5) : > 3


Task 3-5
using above defined boundary curves to generate domain Omega
Enter suffix of file to save coordinates to >> saved x and y coordinates of normal grid
Enter suffix of file to save coordinates to >> saved x and y coordinates of lower boundary resolved grid
```

8.2 Curvebase Class Header - curvebase.h

```
#ifndef _CURVEBASE
#define _CURVEBASE



class Curvebase {

protected:
        double pmin; //lower boundary of curve
        double pmax; //upper boundary of curve
        double mid; //mid point of curve
        bool rev = true; // orientation of the curve, default positive
        double tol; //tolerance

        double length; //length of curve

        virtual double xp(double p) = 0; //returns original x curve coordinates
        virtual double yp(double p) = 0; //returns original ycurve coordinates
        virtual double dxp(double p) = 0; //returns original x curve derivative
        virtual double dyp(double p) = 0; //returns original y curve derivative

        double function(double p); //function to be integrated
        double integrate(double p, double lower); //arc length integral
        double solve(double s); //newton rhapson fixed point iteration solver

        double I1(double a, double b); //help-function for ASI
        double I2(double a, double b); //help-function for ASI


public:

        Curvebase(double p_min,
                        double p_max,
                        bool dir = true,
                        double tolerance = 10e-5);

        Curvebase(); //constructor
        ~Curvebase(); //destructor
        Curvebase(const Curvebase &cb); //copy constructor
        double x(double s); //arc length parametrization
        double y(double s); //arc length parametrization
        virtual void setLength(void); //compute and set length of curve
        void printInfo(void); //print information about curve
        void reverse_orientation(void); //reverse curve orientation
        bool ori(void);


};

#endif
```

8.3 Curvebase Class Sourcecode - curvebase.cpp

```
#include <iostream>
#include <cmath>
#include "curvebase.h"


//constructor
Curvebase::Curvebase(double p_min,
                                        double p_max,
                                        bool dir,
                                        double tolerance) {

        pmin = p_min;
        pmax = p_max;
        mid = (p_min + p_max) / 2.0;
        rev = dir;
        tol = tolerance;;
};

void Curvebase::setLength(void){
        length = abs(integrate(pmax,pmin));
}

//default constructor
Curvebase::Curvebase(){};

//default destructor
Curvebase::~Curvebase(){};

//copy constructor
Curvebase::Curvebase(const Curvebase &cb) {
        pmin = cb.pmin;
        pmax = cb.pmax;
        mid = cb.mid;
        rev = cb.rev;
        tol = cb.tol;
};


//Simpson Adaptive Integration
//Uses Recursion
double Curvebase::integrate(double p, double lower){

        double errest = tol + 1.0;
        double integral1, integral2;


        integral1 = I1(lower,p);
        integral2 = I2(lower,p);

        //estimate error in approximation
        errest = abs(integral1 - integral2);

        //refine interval until error below threshold
        if (errest < 15.0*tol) {
                return integral2;
        } else {
                return (integrate((pmin+p)*0.5, pmin) + integrate(p,(pmin+p)*0.5));
        }

};
```

```
//Helper functions for integrate-function

//helper function for integrate function
double Curvebase::I1(double a_in, double b_in){
                return ((b_in -a_in)/6.0)*(function(a_in)+4*function(0.5*(a_in + b_in)) + function(b_in));
}

//helper function for integrate function
double Curvebase::I2(double a_in, double b_in) {
        double c = (a_in + b_in) * 0.5;
        return I1(a_in,c) + I1(c, b_in);
};

//integrand of integral in curve length computation
//used by integrate function in the adaptive integration
double Curvebase::function(double p) {
        return sqrt( pow(dxp(p),2) + pow(dyp(p),2));
};

//End of Helper functions for integrate-function

//returns x-coordinate of relative position along curve
//relative position is given in interval [0,1]
double Curvebase::x(double s){
        if ( abs(s) > 1.0) {
                std::cout << "ERROR: bad relative position of x-coordinate" << std::endl;
        }
        //get parametrized poisiton from relative position
        double p = solve(s);
        //get actual y-coordinate for parametrized position
        return xp(p);
};

//returns y-coordinate of relative position along curve
//relative position is given in interval [0,1]
double Curvebase::y(double s){
        if ( abs(s) > 1.0) {
                std::cout << "ERROR: bad relative position of x-coordinate" << std::endl;
        }
        //get parametrized poisiton from relative position
        double p = solve(s);
        //get actual y-coordinate for parametrized position
        return yp(p);

};

//Newton Rhapson's method to parametrized value
//corresponding to relative position
double Curvebase::solve(double s) {

        //set initial guess to length of
        //line segment between endpoints
        double p = s*(pmax - pmin);
        double p_new;
        double eps = 1e-10;
        double lower;

        int n_iter = 0;
        const int maxiter = 100;

        //iterate untill error is below tolerance or
```

```
        //100 iterations have been performed
        do {
                p = p_new;
                if (function(p) == 0.0) {
                        std::cout << "Zero Division. Bad Value." << std::endl;
                        p_new = p + eps;
                } else {
                        p_new = p - (integrate(p,pmin) - s*length) / function(p);

                }
                ++n_iter;
        } while ((abs(p-p_new) > tol) && n_iter <= maxiter);

        if (n_iter >= maxiter) {
                std::cout << "ERROR: Convergence not reached." << std::endl;
        }
        return p_new;
};

//reverse orientation of curve
void Curvebase::reverse_orientation(void){
        rev = !rev;
};

//returns orientation of curve
bool Curvebase::ori(void){
        return rev;
}

//prints information about curve
//for inquiry of configuration
void Curvebase::printInfo(void){
        std::cout << "lower boundary : " << pmin << std::endl;
        std::cout << "upper boundary : " << pmax << std::endl;
        std::cout << "total length : " << length << std::endl;
        std::cout << "orientation : " << rev << std::endl;
        std::cout << "x(0.5) : " << x(0.5) << std::endl;
        std::cout << "y(0.5) : " << y(0.5) << std::endl;
        return;
};
```

8.4 Horzline Class Header - hline.h

```
#ifndef _HLINE
#define _HLINE

#include "curvebase.h"

//abstract class for hertical lines where x-coordinate is
//a function of the y-coordinate

class Horzline : public Curvebase {
        protected:

                double xp(double p);//x as function of parametrization variable
                double yp(double p);//y as function of parametrization variable
                double dxp(double p);//derivative of x as function of parametrization variable
                double dyp(double p);//derivative of x as function of parametrization variable

                virtual double yfunc(double p) = 0;//derivative of x-coordinate as function of y-coordinate
                virtual double yfuncd(double p) = 0;//derivative of x-coordinate as function of y-coordinate
        public:
                Horzline(double a, double b, bool dir) : Curvebase(a,b,dir) {}; //use same constructor as for Curvebase
                Horzline(); //default constructor
                ~Horzline();//default destructor
};
#endif
```

8.5 Horzline Class Sourcecode - hline.cpp

```
#include "hline.h"

Horzline::Horzline(){};
Horzline::~Horzline(){};

//x as function of parametrization variable
double Horzline::xp(double p){
        return p;

};

//derivative of x as function of parametrization variable
double Horzline::dxp(double p){
        return 1;
};

//y as function of parametrization variable
double Horzline::yp(double p){
        return yfunc(p);
};

//derivative of y as function of parametrization variable
double Horzline::dyp(double p) {
        return yfuncd(p);
};
```

8.6 Vertline Class Header - vline.h

```
#ifndef _VLINE
#define _VLINE

#include "curvebase.h"

//abstract class for vertical lines where x-coordinate is
//a function of the y-coordinate

class Vertline : public Curvebase {
        protected:

                double xp(double p);//x as function of parametrization variable
                double yp(double p);//y as function of parametrization variable
                double dxp(double p);//derivative of x as function of parametrization variable
                double dyp(double p);//derivative of x as function of parametrization variable

                virtual double xfunc(double p) = 0;//derivative of x-coordinate as function of y-coordinate
                virtual double xfuncd(double p) = 0;//derivative of x-coordinate as function of y-coordinate

        public:
                Vertline(double a, double b, bool dir) : Curvebase(a,b, dir) {}; //use same constructor as for curvebase
                Vertline(); //default constructor
                ~Vertline(); //default destructor
};

#endif
```

8.7 Vertline Class Sourcecode - vline.cpp

```
#include "vline.h"

Vertline::Vertline(){};
Vertline::~Vertline(){};


//x as function of parametrization variable
double Vertline::xp(double p){
        return xfunc(p);
};

//derivative of x as function of parametrization variable
double Vertline::dxp(double p){
        return xfuncd(p);
};

//y as function of parametrization variable
double Vertline::yp(double p){
        return p;
};

//derivative of y as function of parametrization variable
double Vertline::dyp(double p){
        return 1;
};
```

8.8 Domain Class Header - domain.h

```
#ifndef _DOMAIN
#define _DOMAIN

#include "curvebase.h"

//class to represent four sided domains
//allows user to generate grids of desired size

class Domain {
        private:
                Curvebase *sides[4]; //Holder for boundary curves
                double *x_, *y_; //arrays for coordinate vectors, pointers for Dynamic Memory Allocation
                int m_, n_; //number of intervals in vertical (m) and horizontal (n) direction
                int n_points; //total number of points

                double xmap(double r, double s);//x-coordinate mapping from unit square do domain
                double ymap(double r, double s);//y-coordinate mapping from unit square do domain

                bool lower_resolve = false;//use lower-boundary resolution if true

        public:
                Domain(); //Default constructur
                Domain(Curvebase &s1, Curvebase &s2,
                        Curvebase &s3, Curvebase &s4); //constructor when provided four boundary curves
                Domain(const Domain &d); //copy  constructor
                Domain& operator=(const Domain &d); //assignment operator overloading
                ~Domain(); //default destructor

                void check_consistency(void);//assign proper side-identity to boundary curves
                void make_grid (int m, int n);//make a mxn-grid over the domain. Will remove old grid.

                Curvebase * getSide(int s);//reuturns the boundary curve of provided identity
                double sigmaT(double s);//distribution of y-coordinates when refinement is used
                void doLowerResolve(bool a);//will increase resolution of lower boundary if true

                void printCoordinates(void);//print the coordinates as comma separated tuples
                void saveCoordinates(bool user_input);//save coordinates.

};
#endif
```

8.9 Domain Class Sourcecode - domain.cpp

```
#include <iostream>
#include <cstdio>
#include <cmath>

#include "domain.h"

/*

*/

//default constructor
Domain::Domain(){};

//default destructor
Domain::~Domain(){};

//copy constructor
//all private variables including grid will be copied
Domain::Domain(const Domain &d) {
        for (int ii = 0; ii <=3; ii++) {
                sides[ii] = d.sides[ii];
        }
        x_ = d.x_;
        y_ = d.y_;

        m_ = d.m_;
        n_ = d.n_;
        n_points = d.n_points;
        lower_resolve = d.lower_resolve;
};

//assignment operator overloading
//all private variables including grid will be set equal
//right sid object
Domain & Domain::operator=(const Domain &d) {
        if (this == &d) {
                return *this;
        } else {

                for (int ii = 0; ii <=3; ii++) {
                        sides[ii] = d.sides[ii];
                }
                x_ = d.x_;
                y_ = d.y_;

                m_ = d.m_;
                n_ = d.n_;
                n_points = d.n_points;
                lower_resolve = d.lower_resolve;

        }

        return *this;

}

//constructor taking four boundary curves
//representing the four sides of the domain
//boundary curves can be passed in any arbitrary order
Domain::Domain(Curvebase &s1, Curvebase &s2,
```

```
                        Curvebase &s3, Curvebase &s4){


        //assign curves to sides array
        //convention of curve position within sides array
        //is 0-left 1-bottom 2-right 3-top
        //this is adjusted after initial assignation

        sides[0] = &s1; //left side
        sides[1] = &s2; //bottom side
        sides[2] = &s3; //right side
        sides[3] = &s4; //top side

        //adjust boundary curve positioning in sides array
        //as to be consistent with above described convention
        check_consistency();

        //reset grid parameters
        m_ = n_ = 0;
        x_, y_ = nullptr;

};

//returns pointer to boudary curve of sides array
//with the provided index "s"
Curvebase * Domain::getSide(int s) {
        return sides[s];
};

//x-coordinate mapping from unit square to domain
double Domain::xmap(double r, double s) {

        //raise warning if r or s are not in [0,1]
        if (s < 0 || s > 1 || r < 0 || r > 1) {
                std::cout << "x(r,s) : parameters out of bound" << std::endl;
        }

        //transform relative y-coordinate if adjusted
        //resolution should be used
        if (lower_resolve){
                s = sigmaT(s);
        }

        //transfinite interpolation

        //positive terms
        double pos = (1. - r)*sides[0]->x(s) +
                                r*sides[2]->x(s) +
                                (1.-s)*sides[1]->x(r) +
                           s*sides[3]->x(r);

        //negative terms
        double neg = (1.-r)*(1.-s)*sides[0]->x(0.0) +
                                r*(1.-s)*sides[1]->x(1.0) +
                                (1-r)*s*sides[3]->x(0.0) +
                                r*s*sides[2]->x(1.0);

        return pos - neg;
};


double Domain::ymap(double r, double s) {
```

```
        //raise warning if r or s are not in [0,1]
        if (s < 0 || s > 1 || r < 0 || r > 1) {
                std::cout << "y(r,s): parameters out of bound" << std::endl;
        }

        //transform relative y-coordinate if adjusted
        //resolution should be used
        if (lower_resolve){
                s = sigmaT(s);
        }


        //transfinite interpolation

        //positive terms
        double pos = (1. - r)*sides[0]->y(s) +
                                r*sides[2]->y(s) +
                                (1.-s)*sides[1]->y(r) +
                           s*sides[3]->      y(r);

        double neg = (1.-r)*(1.-s)*sides[0]->y(0.0) +
                                r*(1.-s)*sides[1]->y(1.0) +
                                (1-r)*s*sides[3]->y(0.0) +
                                r*s*sides[2]->y(1.0);

        return pos - neg;
};

//function to generate grid over domain
//boundary curves need to be passed before grid
//generation.
void Domain::make_grid(int m, int n){

        //check if grid has already been generated over domain
        //if such. remove old grid
        if (!(m <= 0) && (n <= 0)) {
                std::cout << "Erasing old grid" << std::endl;
                delete [] x_;
                delete [] y_;
        }

        m_ = m; //number of horizontal intervals in grid
        n_ = n; //number of vertical intervals in grid


        double dx = 1.0/(double(m_) -1.0);//horizontal sub-interval length
        double dy = 1.0/(double(n_) -1.0);//vertical sub-interval length

        n_points = m_*n_; //total number of grid-points

        x_ = new double [n_points]; //generate array to store x-coordinates in (dynamic)
        y_ = new double [n_points]; //generate array to store y-coordinates in (dynamic)

        //algebraic grid formation
        for (int ii = 0; ii < n_; ii++){
                for (int jj = 0; jj < m_; jj++){

                        x_[ii*m_ + jj] = xmap(dx*(double)jj, dy*(double)ii);
                        y_[ii*m_ + jj] = ymap(dx*(double)jj, dy*(double)ii);

                }
```

```
        }

}

//function to check that boundary curves
//are properly sorted
//allows for boundary curves to be
//arbitrarily passed to domain upon initialization
void Domain::check_consistency(void){

        int neg[2], pos[2]; //arrays to store index of curve orientations
        int p = 0, n = 0;

        //find which sides are negatively
        //respectively positively oriented
        for(int ii = 0; ii < 4; ii++) {
                if(sides[ii]->ori()){
                        pos[p] = ii;
                        p++;
                } else {
                        neg[n] = ii;
                        n++;
                }
        }

        Domain tmp(*this); //temporary domain to access sides from

        //get index of left and top boundary curve
        //assign to position compatible with grid mapping
        int p_neg = (int)(tmp.getSide(neg[0])->y(0.0) < tmp.getSide(neg[1])->y(0.0));
        sides[0] = tmp.getSide(neg[1-p_neg]);
        sides[3] = tmp.getSide(neg[p_neg]);

        //get index of lower and right boundary curve
        //assign to position compatible with grid mapping
        int p_pos =  (int)(tmp.getSide(pos[0])->x(0.0) < tmp.getSide(pos[1])->x(0.0));
        sides[1] = tmp.getSide(pos[1-p_pos]);
        sides[2] = tmp.getSide(pos[p_pos]);

};

//function used to increase resolution of
//lower boundary curve
double Domain::sigmaT(double sigma) {
        return 1 + tanh(3.0 * (sigma-1.0))/tanh(3.0);
};

//call to activate lower boundary increased resolution
//upon grid generation
void Domain::doLowerResolve(bool a) {
        lower_resolve = a;
}

//function to print coordinates in format "x,y"
void Domain::printCoordinates(void) {
        for (int ii = 0; ii < n_points; ii++)
                {
                 std::cout << x_[ii] << "," << y_[ii] << std::endl;
                }
};

//function that generating two .bin files in a directory res
```

```
//x-coordinate vector and y-coordinates vector of grid
//if user_input is set to true the user will be asked
//to provide a suffix to be used for the file name
void Domain::saveCoordinates(bool user_input = false) {
        std::string outname;

        //catch user specified name
        if (user_input){
                        std::cout << "Enter suffix of file to save coordinates to >> ";
                        std::cin >> outname;
        } else {
                outname = "generated_grid.bin";
        }

        //names of files to be saved
        std::string x_outname = "../res/x_vec_" + outname;
        std::string y_outname = "../res/y_vec_" + outname;

        //save x-coordinate file
        FILE *fp_x;
        fp_x =fopen(x_outname.c_str(),"wb");
        fwrite(x_,sizeof(double),m_*n_,fp_x);
        fclose(fp_x);

        //save y-coordinate file
        FILE *fp_y;
        fp_y =fopen(y_outname.c_str(),"wb");
        fwrite(y_,sizeof(double),m_*n_,fp_y);
        fclose(fp_y);

};
```

8.10 Visualization - visual.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Tue Nov 27 17:06:19 2018

@author: Alma Andersson

"""
import matplotlib.pyplot as plt
import numpy as np
import sys

plt.style.use('seaborn-dark')

if sys.argv[1].lower() in ['--help','-h','help']:
    print(f"script to visualize rendered grid over domain\n")
    print(f"enter name of x-coordinate bin-file as first argument")
    print(f"enter name of y-coordinate bin-file as second argument")
    sys.exit(0)

if sys.argv[1].split('.')[-1] != 'bin' or sys.argv[2].split('.')[-1] != 'bin':
    print(f'Please enter name of two bin files containing x and y coordinates')
    sys.exit(0)

try:
    with open(sys.argv[1],"rb") as fopen:
        x_vec = np.fromfile(fopen,count = -1)

except FileNotFoundError:
    print(f"x-vector file not found")
    sys.exit(0)

try:
    with open(sys.argv[2],"rb") as fopen:
        y_vec = np.fromfile(fopen,count = -1)
except FileNotFoundError:
    print(f"y-vector file not found")
    sys.exit(0)

x_vec = np.array(x_vec)
y_vec = np.array(y_vec)


plt.scatter(x_vec,y_vec, s = 10,zorder = 1,color = 'black')
plt.xlabel('x')
plt.ylabel('y')
plt.title(r'20x50 grid generated for domain $\Omega$')
plt.show()
```