# Program construction in C++ for Scientific Computing - Project 3

Pia Callmer, Jonas Fey

November 2022

The task of this project is to implement a general class for constructing 4-sided domains and grids. In order to do this, we first implement a curvebase class, which can then be used to build a domain class.

## Task 1

In the first task we implemented the non-virtual functions in the `Curvebase` class such as the `double integrate(double p)` function, which returns the arc length integral. In addition, we added the non-virtual functions `funcintegrate, ASI, I_2, I, newton, f`, which are used to calculate the integral. The functions `ASI` with its helper functions, `I_2` and `I` are taken from the first project to approximate the integral. The function `funcintegrate` represents the function $\sqrt{x'(q)^2 + y'(q)^2}$, which needs to be integrated in order to calculate the arclength $l(p)$ given by

$$l(p) = \int_a^p \sqrt{x'(q)^2 + y'(q)^2} dq.$$

The functions `x(s)` and `y(s)` return the $x$ or $y$ values respectively for a given arc length $s \in [0, 1]$. Here, the parameter $p$ is needed to calculate the $x$ or $y$ values with `xp(p)` or `yp(p)`. This parameter $p$ is calculated using newtons method by finding the root of $l(p) - s \cdot l(b)$, which is implemented with the function `f` in our `Curvebase` class. For the newton function the derivatives `dxp` and `dyp` are needed.

The class attributes `pmin` and `pmax` represent the minimal and the maximal value for $p$, which parameterize the curve. For example if we have a straight horizontal boundary line, `pmin` would represent the minimum $x$ value, i.e. where the line starts. The attribute `rev` stands for reverse and indicates the orientation of the curve. This is important when constructing the domain and checking the consistency of the edges. In our implementation we number the edges anti-clockwise starting from the boundary with the lowers $y$ value. If `pmin` is larger than `pmax`, when constructing the curvebase, the orientation is reversed and `rev` is set to true. This influences also in which direction we integrate and evaluate the `Curvebases`.

## Task 2

The grid to be generated consists of four curves, two vertical curves between $y = 0$ and $y = 3$ (at $x = 5$ and $x = -10$ respectively), one horizontal curve going from $x = 5$ to $x = -10$ at height $y = 3$ and one curve with base height $y = 0$ which is defined by the following functions

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

To implement these cases, three classes derived from `Curvebase` were implemented. The class `Yhorizontal` goes from $x_0$ to $x_f$ at height $y_0$ which are the input parameters when creating an object from that class. The previously described functions `x`, `xp`, `dxp`, `y`, `yp` and `dyp` are overwritten as no `newton` method is needed

to calculate $p$. Instead it can be determined by projecting the arc length $s$ onto the line from $x_0$ to $x_f$ with the formula

$$p = p_{min} + s \cdot \text{length}, \quad \texttt{rev=false}$$

or, if the curve is reversed

$$p = p_{max} - s \cdot \text{length}, \quad \texttt{rev=true}.$$

Following that, the $x$ value is equal to the parameter $p$ on the straight line while the $y$ value is constant at $y = y_0$. Therefore, the derivatives are 1 and 0 respectively.

The class `Xvertical` was implemented with the same concept but this time the $x$ coordinate is constant at $x_0$ while the $y$ coordinate goes from $y_0$ to $y_f$.

Finally, the class `Nonconstcurve` was implemented to represent the curve described by equation (1). It uses the described functions `x`, `y`, as well as the `newton` and `ASI` method of its parent class. An object of this class is defined by giving an interval for the $x$ coordinate by defining the parameters $x_0$ and $x_f$. The base height $y_0$ is always assumed to be 0 for this case. Because the function is defined as $f(x)$ ($x$ being the parameter), the functions `xp` and `dxp` simply return $p$ and 1 respectively. The function `yp` returns the result of equation (1) and `dyp` its derivative which was evaluated to be

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

The classes were tested by recreating the Domain as seen in the exercise. A plot of this was made in MATLAB and can be seen in Figure1 as the red outer line.

# Task 3

With the derived `Curvebases` described in task 2, it is now possible to define the domain. For this a new class named `Domain` is needed. This class takes exactly four curvebases as input and creates a domain consisting of those four curves. For this to work, a consistency check is performed during object creation. The consistency check confirms that the ending point of every curve is in close proximity to the starting point of the following curve. This ensures that the domain is well defined. The program will terminate and throw an error, should the consistency check fail.

Finally, the function `generate_grid(int n, int m)` creates a grid of size $n \times m$ in the defined domain. This is done by using the formula derived in the lecture

$$\begin{aligned} \Phi(\xi,\eta) =& \varphi_1(\xi)\Phi(0,\eta) + \varphi_2(\xi)\Phi(1,\eta) \\ &+\varphi_1(\eta)\Phi(\xi,0) + \varphi_2(\eta)\Phi(\xi,1) \\ &-\varphi_1(\xi)\varphi_1(\eta)\Phi(0,0) \\ &-\varphi_1(\xi)\varphi_2(\eta)\Phi(0,1) \\ &-\varphi_2(\xi)\varphi_1(\eta)\Phi(1,0) \\ &-\varphi_2(\xi)\varphi_2(\eta)\Phi(1,1) \end{aligned}$$

where $\xi_i = ih_1$ with $h_1 = \frac{1}{n}$ as the step size in $\xi$ direction and $\eta_j = jh_2$ $h_2 = \frac{1}{m}$ as the step size in $\eta$ direction. The interpolation functions $\varphi$ are defined as $\varphi_1(s) = 1 - s$ and $\varphi_1(s) = s$. The expressions $\Phi(0,\eta)$, $\Phi(1,\eta)$, $\Phi(\xi,0)$, ... are all located on the border domain and can be evaluated using the `x` and `y` functions of the corresponding `Curvebases`.

# Task 4

Finally, the generated grid described in task 3 is saved to a `.bin` file and exported to MATLAB where the grid and the domain are plotted to evaluate the solution. Figure 1 shows in red the outer domain border defined by the four `Curvebases` and in blue the generated grid points by the `generate_grid` function with $n = 50$ and $m = 20$. It can be seen, that all grid points are inside the domain or on top of its border. Also, the grid points follow the shape of the lower non-constant curve to create an adequate spacing between each grid point which indicates a suitable grid.

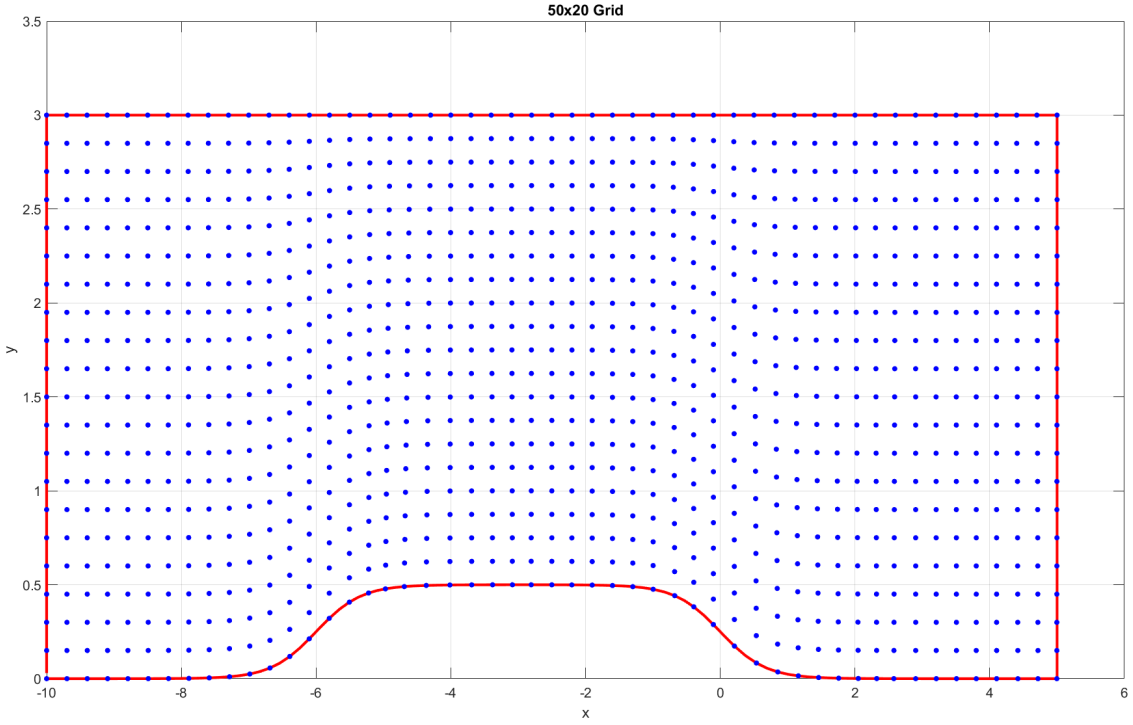


Figure 1: Final grid using $50 \times 20$ elements. The blue points indicate the grid points and the red line shows the original curves defining the domain.

# A Appendix – Code files

## A.1 main.cpp

```
#include <iostream>
#include "curvebase.hpp"
#include "xvertical.hpp"
#include "yhorizontal.hpp"
#include "nonconstcurve.hpp"
#include "domain.hpp"
#include<cstdio>

using namespace std;



int main()
{

    // Generate four curves that define the grid
    Nonconstcurve s1 = Nonconstcurve(-10.0, 5.0);
    Xvertical s2 = Xvertical(0.0, 3.0, 5.0);
    Yhorizontal s3 = Yhorizontal(5.0, -10.0, 3.0);
    Xvertical s4 = Xvertical(3.0, 0.0, -10.0);

    // Generate domain
    Domain mydomain = Domain(s1,s2,s3,s4);

    // Generate grid with size of 50x20
    mydomain.generate_grid(50,20);


    cout << "------------" << endl;


    // Save the grid to the file 'grid_out.bin'
    int counter = 0;
    int len = (mydomain.M_()+1)*(mydomain.N_()+1);
    double grid[2*len];
    for(int i=0; i<2*len;i+=2){
        grid[i] = mydomain.X_()[counter];
        grid[i+1] = mydomain.Y_()[counter];
        counter+=1;
    }

    FILE *fp;
    fp =fopen("grid_out.bin","wb");
    fwrite(grid,sizeof(double),2*len,fp);
    fclose(fp);



    // Save the curvebases to the file 'curve_out.bin'
    double stepsize = 0;
```

```
    len = (4*2*100);
    double curves[2*len];

    curves[200] = 1.0;

    for(int i=0; i<2*100;i+=2){
        curves[i] = mydomain.Sides()[0]->x(stepsize);
        curves[i+1] = mydomain.Sides()[0]->y(stepsize);

        curves[i+200] = mydomain.Sides()[1]->x(stepsize);
        curves[i+1+200] = mydomain.Sides()[1]->y(stepsize);

        curves[i+400] = mydomain.Sides()[2]->x(stepsize);
        curves[i+1+400] = mydomain.Sides()[2]->y(stepsize);

        curves[i+600] = mydomain.Sides()[3]->x(stepsize);
        curves[i+1+600] = mydomain.Sides()[3]->y(stepsize);

        stepsize+=(1.0/(100.0));
    }

    FILE *curvefile;
    curvefile =fopen("curve_out.bin","wb");
    fwrite(curves,sizeof(double),2*len,curvefile);
    fclose(curvefile);

    cout << "Grid␣saved␣to␣'grid_out.bin'!" << endl;


    return 0;
}
```

## A.2 Curvebase.hpp

```
#ifndef CURVEBASE_HPP
#define CURVEBASE_HPP

typedef double(*FunctionPointer)(double);

class Curvebase{
protected:
    double pmin;  // a --> start value for p of the parameterised curve
    double pmax;  // b --> end value for p of the parameterised curve
    bool rev; // Boolean that indicates if the curve is reversed
    double length; // Length of the curve

    // ....

    virtual double xp(double p) = 0; // Parameterised curve in p. Returns
       ↪ x-value for given p
    virtual double yp(double p) = 0; // Parameterised curve in p. Returns
       ↪ y-value for given p
    virtual double dxp(double p) = 0; // Parameterised curve in p. Returns
       ↪  derivative in x for given p
    virtual double dyp(double p) = 0; // Parameterised curve in p. Returns
       ↪  derivative in y for given p

    double integrate(double p); // arc length integral
    double funcintegrate(double q); // function to integrate using ASI

    double ASI(double a, double  b, double tol); // ASI numerical
       ↪ integration
    double I_2(double a, double b); // Help function for ASI
    double I(double a, double b); // Help function for ASI

    double newton(double s, double p0, double tol); // Newton method
    double f(double p, double s); // Function to find a root on using
       ↪ newton --> the root gives the parameter p based on given s.


    // .....

public:
    Curvebase(); // Constructor
    ~Curvebase(); // Destructor

    double x(double s); // Returns x value of curve for given arc length
       ↪ parameter s in [0,1]
    virtual double y(double s); // Returns y value of curve for given arc
       ↪ length parameter s in [0,1]

    double Pmin() const {return pmin;}; // Returns the minimum value p of
       ↪ the parameterised curve
    double Pmax() const {return pmax;}; // Returns the maximum value p of
       ↪ the parameterised curve
    bool Rev() const {return rev;}; // Returns if the curve is reversed or
```

```
        ↪  not
    double Length() const {return length;}; // Returns the length of the
        ↪ curve

    void printCurve(); // Prints the relevant information of the curve
        ↪ using cout

    // .....

};

#endif
```

### A.3 Curvebase.cpp

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

#include "curvebase.hpp"
using namespace std;


//Constructor
Curvebase::Curvebase(){}

// Destructor
Curvebase::~Curvebase(){}


// Integrate function that integrates funcintegrate from pmin to p using
   ↪ the ASI function
double Curvebase::integrate(double p){
    return ASI(pmin, p,0.01);
}


// Function to integrate / derivative
double Curvebase::funcintegrate(double q){
    return sqrt(pow(dxp(q),2.0) + pow(dyp(q),2.0));
}

// Returns x value for given arc length s in [0,1]
double Curvebase::x(double s){
    double p;
    // Use newton method to find parameter p for given s. Use 1-s for a
       ↪ reversed curve
    if(rev) p = newton(1.0-s,(pmin+pmax)/2.0,0.01);
    else p = newton(s,(pmin+pmax)/2.0,0.01);
    // Return the x value for given parameter p of the parameterised curve
    return xp(p);
}

// Returns y value for given arc length s in [0,1]
double Curvebase::y(double s){
    double p;
    // Use newton method to find parameter p for given s. Use 1-s for a
       ↪ reversed curve
    if(rev) p = newton(1.0-s,(pmin+pmax)/2.0,0.01);
    else p = newton(s,(pmin+pmax)/2.0,0.01);
    // Return the y value for given parameter p of the parameterised curve
    return yp(p);
}


// Newton method with initial guess p0 and tolerance tol
double Curvebase::newton(double s, double p0, double tol){
    double res=p0;
```

```
    while(abs(f(res,s)) > tol){
        // Using Newton scheme to find root
        res = res - f(res,s)/funcintegrate(res);
    }
    return res;
}

// Function l(p) - s * l(pmax) that is used to find parameter p
double Curvebase::f(double p, double s){
    return integrate(p) - s*length;
}



// Use Simpsons rule to estimate Integral value
double Curvebase::I(double a, double b){
    return ((b-a)/6)*(funcintegrate(a) +4*funcintegrate((a+b)/2)+
       ↪ funcintegrate(b));

}

// Use I2  to decrease error value for Integral approximation
double Curvebase::I_2(double a, double b){
    double gamma = (a+b)/2;
    return I(a, gamma) + I(gamma,b);
}

// Adaptive Simpson Estimation
double Curvebase::ASI(double a, double  b, double tol){
    double I1 =  I(a, b);
    double I2 = I_2(a, b);
    double errest = std::abs(I1-I2);
    if (errest < 15*tol){
        return I2;
    }
    return ASI(a, (a+b)/2, tol/2) + ASI((a+b)/2, b, tol/2);
}


// Print parameters using cout
void Curvebase::printCurve(){
    std::cout << pmin << pmax << std::endl;
    std::cout << rev << length << std::endl;
}
```

## A.4 Xvertical.hpp

```
#include "curvebase.hpp"
#ifndef XVERTICAL_HPP
#define XVERTICAL_HPP

class Xvertical: public Curvebase{
protected:
    double xp(double p); // Parameterised curve in p. Returns x-value for
       ↪ given p
    double yp(double p); // Parameterised curve in p. Returns y-value for
       ↪ given p
    double dxp(double p); // Parameterised curve in p. Returns derivative
       ↪ in x for given p
    double dyp(double p); // Parameterised curve in p. Returns derivative
       ↪ in y for given p


private:
    double xconst; // Parameter that stores the constant x-value of the
       ↪ vertical line

public:
    Xvertical(double y0, double yf, double x0); // Constructor for
       ↪ vertical line going from y0 to yf at x-value = x0
    Xvertical& operator=(const Xvertical& curve); // Copy assignement
    ~Xvertical(){}; // Destructor
    double x(double s); // Returns x value of curve for given arc length
       ↪ parameter s in [0,1]
    double y(double s); // Returns y value of curve for given arc length
       ↪ parameter s in [0,1]

    double Xconst() const {return xconst;}; // Returns the constant x-
       ↪ value of the vertical line

    void printCurve();
    // .....

};

#endif
```

### A.5 Xvertical.cpp

```
#include <iostream>
#include <cmath>
#include "curvebase.hpp"
#include "xvertical.hpp"


// Constructor for vertical line
Xvertical::Xvertical(double y0, double yf, double x0){

    xconst = x0;

    // Check if y0 or yf is lower and adjust pmin, pmax and rev
       ↪ accordingly
    if(y0 < yf){
        rev = false;
        pmin = y0;
        pmax = yf;
    } else {
        rev = true;
        pmin = yf;
        pmax = y0;
    }


    // Length of the curve
    length = abs(yf - y0);
}


// Assign operator
Xvertical& Xvertical::operator=(const Xvertical& curve){
    if(this != &curve){
        pmin = curve.Pmin();
        pmax = curve.Pmax();
        rev = curve.Rev();
        length = curve.Length();
        xconst = curve.Xconst();
    }
    return *this;
}



// Since it is a straight vertical line, the x-value is constant
double Xvertical::xp(double p){
    return xconst;
}

// Following xp, its derivative is 0
double Xvertical::dxp(double p){
    return 0;
```

```
}

// Since it is a straight vertical line, the y-value is equal to p
double Xvertical::yp(double p){
    return p;
}

// Following yp, its derivative is const = 1
double Xvertical::dyp(double p){
    return 1;
}

double Xvertical::x(double s){
    double p;
    // Since it is a straight line, p can be obtained analytically using
       ↪ the length --> no newton method needed
    if (rev) p = pmax - s*length;
    else p = pmin + s*length;
    return xp(p);
}

double Xvertical::y(double s){
    double p;
    // Since it is a straight line, p can be obtained analytically using
       ↪ the length --> no newton method needed
    if (rev) p = pmax - s*length;
    else p = pmin + s*length;
    return yp(p);
}

void Xvertical::printCurve(){
    std::cout << pmin << pmax << std::endl;
    std::cout << rev << length << std::endl;
}
```

## A.6 Yhorizontal.hpp

```
#include "curvebase.hpp"
#ifndef YHORIZONTAL_HPP
#define YHORIZONTAL_HPP

class Yhorizontal: public Curvebase{
protected:
    double xp(double p); // Parameterised curve in p. Returns x-value for
       ↪ given p
    double yp(double p); // Parameterised curve in p. Returns y-value for
       ↪ given p
    double dxp(double p); // Parameterised curve in p. Returns derivative
       ↪ in x for given p
    double dyp(double p); // Parameterised curve in p. Returns derivative
       ↪ in y for given p


private:
    double yconst; // Parameter that stores the constant y-value of the
       ↪ horizontal line

public:
    Yhorizontal(double x0, double xf, double y0); // Constructor for
       ↪ horizontal line going from x0 to xf at y-value = y0
    Yhorizontal& operator=(const Yhorizontal& curve); // Copy assignement
    ~Yhorizontal(){}; // Destructor
    double x(double s); // Returns x value of curve for given arc length
       ↪ parameter s in [0,1]
    double y(double s); // Returns y value of curve for given arc length
       ↪ parameter s in [0,1]

    double Yconst() const {return yconst;}; // Returns the constant y-
       ↪ value of the horizontal line

    // .....

};

#endif
```

### A.7 Yhorizontal.cpp

```
#include <iostream>
#include <cmath>
#include "curvebase.hpp"
#include "yhorizontal.hpp"


// Constructor for horizontal line
Yhorizontal::Yhorizontal(double x0, double xf, double y0){

    yconst = y0;

    // Check if x0 or xf is lower and adjust pmin, pmax and rev
       ↪ accordingly
    if(x0 < xf){
        rev=false;
        pmin = x0;
        pmax = xf;
    } else {
        rev=true;
        pmin = xf;
        pmax = x0;
    }

    // Length of curve
    length = abs(xf - x0);
}

// Assign operator
Yhorizontal& Yhorizontal::operator=(const Yhorizontal& curve){
    if(this != &curve){
        pmin = curve.Pmin();
        pmax = curve.Pmax();
        rev = curve.Rev();
        length = curve.Length();
        yconst = curve.Yconst();
    }
    return *this;
}

// Since it is a straight horizontal line, the x-value is equal to the
   ↪ parameter p
double Yhorizontal::xp(double p){
    return p;
}

// Following xp, its derivative is constant = 1
double Yhorizontal::dxp(double p){
    return 1;
}

// Since it is a straight horizontal line, the y-value is constant
double Yhorizontal::yp(double p){
```

```
    return yconst;
}

// Following yp, its derivative is 0
double Yhorizontal::dyp(double p){
    return 0;
}

// Return x-value for given arc length s in [0,1]
double Yhorizontal::x(double s){
    double p;
    // Since it is a straight line, p can be obtained analytically using
       ↪ the length --> no newton method needed
    if (rev) p = pmax - s*length;
    else p = pmin + s*length;
    return xp(p);
}

// Return y-value for given arc length s in [0,1]
double Yhorizontal::y(double s){
    double p;
    // Since it is a straight line, p can be obtained analytically using
       ↪ the length --> no newton method needed
    if (rev) p = pmax - s*length;
    else p = pmin + s*length;
    return yp(p);
}
```

### A.8 Nonconstcurve.hpp

```
#include "curvebase.hpp"
#ifndef NONCONSTCURVE_HPP
#define NONCONSTCURVE_HPP

class Nonconstcurve: public Curvebase{
protected:
    double xp(double p); // Parameterised curve in p. Returns x-value for
       ↪ given p
    double yp(double p); // Parameterised curve in p. Returns y-value for
       ↪ given p
    double dxp(double p); // Parameterised curve in p. Returns derivative
       ↪ in x for given p
    double dyp(double p); // Parameterised curve in p. Returns derivative
       ↪ in y for given p


public:
    Nonconstcurve(double x0, double xf); // Constructor of a curve
       ↪ described by the two equations given by the exercise. The y-
       ↪ translations is always y0=0
    Nonconstcurve& operator=(const Nonconstcurve& curve); // Copy
       ↪ assignement
    ~Nonconstcurve(){}; // Destructor


    // .....

};

#endif
```

### A.9 Nonconstcurve.cpp

```
#include <iostream>
#include <cmath>
#include "curvebase.hpp"
#include "nonconstcurve.hpp"

// Constructor
Nonconstcurve::Nonconstcurve(double x0, double xf){
    pmin = x0;
    pmax = xf;

    // Check if x0 or xf is lower and adjust pmin, pmax and rev
       ↪ accordingly
    if(x0 < xf){
        pmin = x0;
        pmax = xf;
        rev = false;
    } else {
        pmin = xf;
        pmax = x0;
        rev = true;
    }

    // Length of the curve
    length = integrate(pmax);
}

// Since the curve described by the exercise changes in x, its x value is
   ↪ equal to the parameter p
double Nonconstcurve::xp(double p){
    return p;
}

// Following xp, its derivative is const = 1
double Nonconstcurve::dxp(double p){
    return 1;
}

// The y-value is given by the equations as shown in the exercise
double Nonconstcurve::yp(double p){
    if(p < -3){
        return 0.5 * ((1)/(1+exp(-3*(p+6)))); // first equation
    } else{
        return 0.5 * (1/(1+exp(3*p))); // second equation
    }
}

// Following yp, the derivatives of the equations can be implemented
double Nonconstcurve::dyp(double p){
    if(p < -3){
        return (1.5 * exp(-3*(p+6)))/((exp(-3*(p+6))+1) * (exp(-3*(p+6))
           ↪ +1)); // first derivative of equation
    } else{
```

```
        return (-1.5 * exp(3*p))/((exp(3*p)+1) * (exp(3*p)+1)); // second
            ↪ derivative of equation
    }
}
```

### A.10 Domain.hpp

```
#include "curvebase.hpp"
#ifndef DOMAIN_HPP
#define DOMAIN_HPP


class Domain{

private:
    // Four curvebases defining the border of the domain
    Curvebase* sides[4];

    // Double arrays that will store the coordinates of the grid points
    double* x_ = nullptr;
    double* y_ = nullptr;


    // Size of the grid
    int n_;
    int m_;

    // Checks the consistency of the border of the domain
    bool check_consistency();

    // Help interpolation functions for generate_grid
    double phi1(double s){return 1-s;};
    double phi2(double s){return s;};

public:
    // Constructor
    Domain(Curvebase& s1, Curvebase& s2, Curvebase& s3, Curvebase& s4);
    // Destructor
    ~Domain();
    // Generate grid in domain with size nxm
    void generate_grid(int n, int m);
    // Prints the grid using cout
    void printGrid();

    // Getter Functions
    Curvebase** Sides(){return sides;};

    double* X_(){return x_;};
    double* Y_(){return y_;};

    int N_(){return n_;};
    int M_(){return m_;};
};

#endif
```

### A.11 Domain.cpp

```
# include "domain.hpp"
# include <iostream>
# include <cmath>

using namespace std;

// Constructor
Domain::Domain(Curvebase& s1, Curvebase& s2, Curvebase& s3, Curvebase& s4)
    ↪ {

    // Four curvebases make up the Domain
    sides[0] = &s1;
    sides[1] = &s2;
    sides[2] = &s3;
    sides[3] = &s4;

    // Check if the curvebases define a domain
    if(check_consistency()){
        cout << ("Domain␣successfully␣generated!") << endl;
    } else {
        sides[0] = nullptr;
        sides[1] = nullptr;
        sides[2] = nullptr;
        sides[3] = nullptr;
        cout << "Domain␣is␣not␣consistend!" << endl;
        exit(1);
    }
}


// Destructor
Domain::~Domain(){
    delete[] x_;
    delete[] y_;
}

// Consistency check
bool Domain::check_consistency(){
    for(int i = 0; i < 4; i++){
        Curvebase* current = sides[i];
        Curvebase* next = sides[(i+1)%4];

        // If current endpoint is not close enough to the next start point
            ↪ , the domain is not well defined
        if(!(abs(current->x(1) - next->x(0)) < 0.1 && abs(current->y(1) -
            ↪ next->y(0)) < 0.1)){
            return false;
        }
    }

    return true;
}
```

```
// Generate grid
void Domain::generate_grid(int n, int m){
    if(m < 1 || n < 1) exit(1);
    if(m_ > 0){
        delete[] x_; delete[] y_;
    }

    // Define sizes of grid points vectors
    x_ = new double[(m+1) * (n+1)];
    y_ = new double[(m+1) * (n+1)];
    m_ = m;
    n_ = n;

    // Step size between grid points
    double h1 = 1.0 / n;
    double h2 = 1.0 / m;

    for(int i = 0; i <= n; ++i){
        for(int j = 0; j <= m; ++j){

            // x coordinate of grid points
            x_[i + j*(n+1)] = phi1(i*h1) * sides[3]->x(1-j*h2) + phi2(i*h1
                ↪ ) * sides[1]->x(j*h2)
                            +phi1(j*h2) * sides[0]->x(i*h1) + phi2(j*h2)
                                ↪ * sides[2]->x(1-i*h1)
                            -phi1(i*h1) * phi1(j*h2) * sides[0]->x(0)  //
                                ↪  Lower left corner point
                            -phi1(i*h1) * phi2(j*h2) * sides[3]->x(0)  //
                                ↪  Upper left corner point
                            -phi2(i*h1) * phi1(j*h2) * sides[1]->x(0)  //
                                ↪  Lower right corner point
                            -phi2(i*h1) * phi2(j*h2) * sides[2]->x(0); //
                                ↪  Upper right corner point


            // y coordinate of grid points
            y_[i + j*(n+1)] = phi1(i*h1) * sides[3]->y(1-j*h2) + phi2(i*h1
                ↪ ) * sides[1]->y(j*h2)
                            +phi1(j*h2) * sides[0]->y(i*h1) + phi2(j*h2)
                                ↪ * sides[2]->y(1-i*h1)
                            -phi1(i*h1) * phi1(j*h2) * sides[0]->y(0)  //
                                ↪  Lower left corner point
                            -phi1(i*h1) * phi2(j*h2) * sides[3]->y(0)  //
                                ↪  Upper left corner point
                            -phi2(i*h1) * phi1(j*h2) * sides[1]->y(0)  //
                                ↪  Lower right corner point
                            -phi2(i*h1) * phi2(j*h2) * sides[2]->y(0); //
                                ↪  Upper right corner point
        }
    }

    cout << "Grid␣generated!" << endl;
}
```

```
// Print function
void Domain::printGrid(){
    cout << "x:" << endl;
    for(int i = 0; i <= n_; ++i){
        for(int j = 0; j <= m_; ++j){
            cout << x_[i + j*(n_+1)] << "␣␣␣";
        }
        cout << endl;
    }

    cout << endl << "y:" << endl;

    for(int i = 0; i <= n_; ++i){
        for(int j = 0; j <= m_; ++j){
            cout << y_[i + j*(n_+1)] << "␣␣␣";
        }
        cout << endl;
    }
}
```

### A.12 Plotting.m

```
file1 = fopen("grid_out.bin");
A=fread(file1,[2,51*21],'double');

file2 = fopen("curve_out.bin");
B=fread(file2,[2,400],'double');


figure(1);
plot(B(1,:), B(2,:), "-r", "LineWidth",2);
hold on;
scatter(A(1,:), A(2,:),15,"blue","filled");
grid on;
xlabel("x");
ylabel("y");
title("50x20 Grid");
```