Project 3
SF2565 Program construction in C++ for Scientific Computing

Caroline Eriksson Sophie Malmliden

November 2019

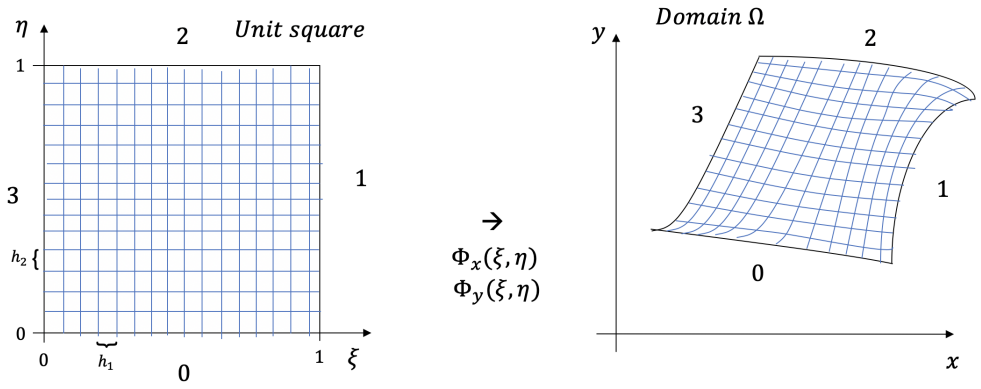

Figure 1: Transfinite Grid Generation

Introduction

The goal of this project is to implement a general class for modelling domains $\Omega \subset \mathbb{R}^2$ and structured grids on them. The domains considered are foursided and can be naturally mapped onto the unit square. The one to one mapping from the boundary of the unit square onto the boundary of Ω is denoted: $\Phi : [0,1]^2 \to \Omega$. The unit square is to have the following uniform grid for given m and n:

$$\xi_i = ih_1, \quad h_1 = 1/n, \quad i = 0, ..., n,$$

$$\eta_j = jh_2, \quad h_2 = 1/m, \quad j = 0, ..., m.$$

A structured grid on Ω can then simply be obtained via:

$$x_{ij} = \Phi_x(\xi_i, \eta_j), \quad y_{ij} = \Phi_y(\xi_i, \eta_j), \quad i = 0, ..., n, \quad j = 0, ..., m.$$

In order to construct $\Phi_x(\xi, \eta)$ and $\Phi_y(\xi, \eta)$ we need to present linear interpolation functions:

$$\varphi_0(s) = 1 - s, \quad \varphi_1(s) = s$$

Using the algebraic grid generation formula, under the assumption that $\Phi_x(\xi,\eta)$ and $\Phi_y(\xi,\eta)$ are known on the sides of the domain, we achieve the entire grid generation corresponding to the grid for the unit square:

$$\Phi_x(\xi,\eta) = \varphi_0(\xi)\Phi_x(0,\eta) + \varphi_1(\xi)\Phi_x(1,\eta) + \varphi_0(\eta)\Phi_x(\xi,0) + \varphi_1(\eta)\Phi_x(\xi,1)+$$

$$-\varphi_0(\xi)\varphi_0(\eta)\Phi_x(0,0)-\varphi_1(\xi)\varphi_0(\eta)\Phi_x(1,0)-\varphi_0(\xi)\varphi_1(\eta)\Phi_x(0,1)-\varphi_1(\xi)\varphi_1(\eta)\Phi_x(1,1)$$

$$\Phi_y(\xi,\eta) = \varphi_0(\xi)\Phi_y(0,\eta) + \varphi_1(\xi)\Phi_y(1,\eta) + \varphi_0(\eta)\Phi_y(\xi,0) + \varphi_1(\eta)\Phi_y(\xi,1)+$$

$$-\varphi_0(\xi)\varphi_0(\eta)\Phi_y(0,0)-\varphi_1(\xi)\varphi_0(\eta)\Phi_y(1,0)-\varphi_0(\xi)\varphi_1(\eta)\Phi_y(0,1)-\varphi_1(\xi)\varphi_1(\eta)\Phi_y(1,1)$$

The first two terms of the $\Phi_x(\xi,\eta)$ and $\Phi_y(\xi,\eta)$ are the interpolation between the sides $\xi = 0$ and $\xi = 1$, the next two terms are the interpolation between the sides $\eta = 0$ and $\eta = 1$. The four remaining terms are terms for correction of the corners.

Task 1

The goal of this task is to complete the class **Curvebase** by writing the non-virtual functions and adding more variables and functions, if necessary.

The following skeleton of an abstract base class is to be considered:

```
    class Curvebase {
    protected:
double pmin; // minimal value for p
double pmax; // maximal value for p
bool rev; // orientation of the curve
double length;
.....
virtual double xp(double p) = 0;
virtual double yp(double p) = 0;
virtual double dxp(double p) = 0;
virtual double dyp(double p) = 0;
double integrate(double p); //arc length integral
.....
public:
Curvebase(); //constructor
double x(double s); //arc length parametrization
double y(double s); //arc length parametrization
.......
};
```

This skeleton will be saved in a header file called **Curvebase**, see Appendix 1. for **Curvebase** header file. The variable, d for determining the position of a line in xy-coordinate system as well as tol, the tolerance for the Adaptive Simpson's method are added to the **Curvebase** header file. The functions added for the class are a copy constructor, a destructor, and integration functions for the Adaptive Simpson's method for numerical integration. These functions, as well as the ones from the skeleton are constructed in a corresponding source file, see Appendix 2. for **Curvebase** source file.

For this entire project the variable s will be used as a parameter for the boundary lines in the unit square and thus $s \in [0, 1]$ and p will be the used parameter for the boundary lines of the Domain Ω and thus $p \in [pmin, pmax]$. The function *integrate* is created as a function computing the arc length of a curve, which is the distance between two points along a section of a curve and can be computed from the following formula:

$$l(p) = \int_{pmin}^{p} (dxp(q)^2 + dyp(p)^2)^{1/2} dq$$

Here, q is used as the parameter of the curve section from $pmin$ to p. By setting p to $pmax$ one can compute the entire length of one boundary curve. The function *integrate* will also be constructed using the Adaptive Simpson's method, from Homework 1, Task 2. The functions I_1 and I_2 are helpfunctins from the Adaptive Simpson's method.

The arc length parametrization functions $x(s)$ and $y(s)$ are, using a given value s, to find the corresponding values for $p, xp(p), yp(p)$. The parametrization functions are accomplishing this using Newton's method, a numerical root-finding algorithm, i.e. approximating the roots of a function. Newton's method can be written:

$$p = p0 - \frac{f(p)}{f'(p)}$$

In this case, that function will be: $f(p) = l(p) - s \cdot l(pmax)$, and thus p will be found for $f(p) = 0$, which means finding the p that, for a given boundary curve from the domain Ω corresponding to a boundary curve from the unit square, will correspond to a given s. p_0 is in this case an initial guess for p. The derivative can be computed as:

$$f'(p) = l'(p) = (dxp(p)^2 + dyp(p)^2)^{1/2}$$

In the code, $f(p)$ will simply be written as: $f = integrate(pmin, p0, tol) - s * length$ and $f'(p)$ will be written as $df = f'(p) = l'(p) = (dxp(p)^2 + dyp(p)^2)^{1/2}$ and added to the **Curvebase** class. Thus the Newton method will be written as: $p = p0 - f/df$ and $p0$ will be updated with each iteration until the error gets small enough. $length$ will be determined from derived classes further on in the project as the length of a boundary curve of the domain Ω and can be computed with the arc length integral function $integrate$. Also dxp, dyp, df will be determined from derived classes.

Task 2

The goal of this task is to generate a grid on the, for this task, given domain and to derive the classes from the base class, **Curvebase**, needed to represent the boundary curves of the domain. For this exercise, three derived classes are constructed, one class called **line_v** for creating the two vertical boundary lines, one class called **line_h** for creating the horizontal boundary line on top of the domain and one class called **lower_curve** for creating the boundary curve at the bottom of the domain. See Appendix 3. to 8. for the header files and source files for these derived classes.

In this task we also start with the construction of the **main** file for this project, in which we declare the coordinates for the boundary lines of the domain given $(pmin, pmax)$, as well as their position (d). From **main** the boundary lines are created using the derived classes. See Appendix 9. for the **main** file.

In the derived classes the length of each line is computed as $pmax - pmin$ and the length of the curve with arc length integration. Since the horizontal boundary line on top of the domain, only goes in x-direction, $xp(p)$ can be set to p, which leads dxp to be 1 and $yp(p)$ can be set to the y-position of the line, d, which leads dyp to be 0 in **line_h**. Similarily, since the vertical boundary lines only are going in y-direction $yp(p)$ can be set to p, which leads dyp to be 1 and $xp(p)$ can be set to the x-position of the line, d, which leads dyp to be 0 in **line_v**. df is clearly set to 1 for both these cases, since:

$$df = (dxp(p)^2 + dyp(p)^2)^{1/2}$$

But for **lower_curve**, df will be set simply as the function above. For this curve, xp is p, which leads dxp to be 1 and yp is to be set as the given function for the curve from the task description, but with x set to p. Furthermore dyp is simply to be set to the derivative of that given function with respect to p.

For simplicity, in this project, especially in the code, the four boundary curves of the given domain will be named boundary curve 0, 1, 2 and 3. Where boundary curve 0 is the lower boundary curve. Boundary curve 1 is the vertical boundary curve to the right of the domain. Boundary curve 2 is the horizontal boundary curve on top of the domain, and boundary curve 3 is the vertical boundary curve to the left of the domain. For numeration, see also Figure 1 in introduction part.

Task 3

The goal of this task is to design a **Domain** class containing four boundary curves of type **Curvebase** and having the capability for generating a grid on the domain from Task 2. One should also write a **main** program which generates the grid.

For this task, the **main** file will be further constructed such that the grid for the domain from Task 2 will be generated through **main** using the **Domain** class which contains four boundary curves of type **Curvebase**. These four boundary curves will for this task be the four boundary curves created in Task 2, which are denoted: $curve\_0$, $line\_v1$, $line\_h2$, $line\_v3$. The grid to be generated is to have 50 grid points in x-direction and 20 grid points in y-direction. Thus the number of grid intervals, see introduction for grid of unit square, will be $n = 49$ intervals in x-direction and $n = 19$ intervals in y-direction. The stepsizes can be computed accordingly: $h_1 = 1/n, \quad h_2 = 1/m$. Further s will then be given as: $s = ih_1, \quad i = 0, ..., n$ and $s = jh_2, \quad j = 0, ..., m$. The grid for the unit square is already decided and now s will be used for the arc length parametrization functions $x(s), y(s)$ which will translate the grid points on the boundary of the unit square into grid points on the boundary of the given domain. The boundary curves for the domain are already given and four vectors are defined for storing the x values and four vectors for storing the y values for the grid point coordinates for each grid point on the boundary curves. These vectors will be filled with values using the arc length parametrization functions $x(s), y(s)$. The remaining grid points inside of the domain are computed using the algebraic grid generation formula, see introduction. These steps for generating the grid on the given domain are put in a function denoted *grid.generation*. See Appendix 10. for the **Domain** header file, and Appendix 11. for the **Domain** source file.

Task 4

The goal of this task is to write the grid to a file. The function added to the **Domain** class is denoted $writetofile()$ and is storing the x and y coordinates for the grid points into files using ofstream. The coordinates for the grid points on the boundary curves are also stored in separate files to be able to display them differently. The generated grid is plotted in Matlab. The grid points on the boundary are plotted with black stars and the interpolated interior grid points are plotted with black dots. The grid lines connecting the grid points are plotted in the color magenta. See Appendix 12 for the Matlab file.

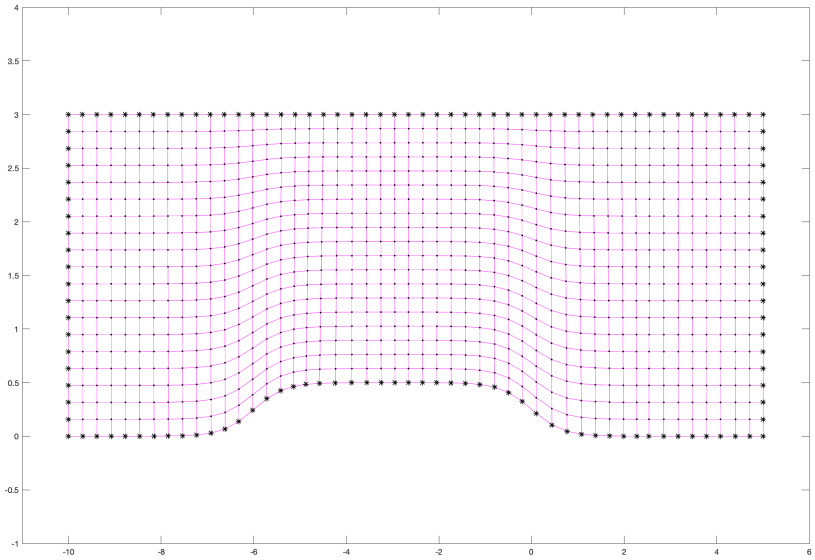

Figure 2: Generation of Grid for given domain

Appendix 1: Curvebase Header File

```
#ifndef CURVEBASE
#define CURVEBASE
#include <cmath>
class Curvebase
{
    friend int main();
protected:
    double pmin;                // Minimal value for p
    double pmax;                // Maximal value for p
    double length;
    double d;
    double tol = pow(10, -2);
    virtual double xp(double p) = 0;
    virtual double yp(double p) = 0;
    virtual double dxp(double p) = 0;
    virtual double dyp(double p) = 0;
    // Arc length integral
    double integrate(double pmin, double pmax, double tol);
    virtual double df(double) = 0;
public:
    Curvebase();                    // Constructor
    ~Curvebase();                   // Destructor
    Curvebase(const Curvebase& object); // Copy constructor
    double x(double s);             // Arc length parametrization
    double y(double s);             // Arc length parametrization
    double I_1(double, double);     // Integral used in ASI
    double I_2(double, double);     // Integral used in ASI
};
#endif
```

Appendix 2: Curvebase Source File

```
#include <cmath>
#include <list>
#include <iostream>
#include "Curvebase.h"
using namespace std;
Curvebase::Curvebase() {};
Curvebase::Curvebase(const Curvebase& object)
{
    pmin = object.pmin, pmax = object.pmax;
    length = object.length, d = object.d;
};
Curvebase::~Curvebase() {};
double Curvebase::x(double s)
{
    double p0 = 0, error = 1, toln = 1e-4, p, f;
    int iter = 0, iter_max = 500;
    while (error > toln && iter < iter_max)
    {
        f = integrate(pmin, p0, tol) - s * length;
        p = p0 - f / df(p0);
        error = fabs(p - p0);
        p0 = p;
        iter += 1;
    }
    if (iter == iter_max)
    {
        cout << "Solution not yet converged to desired"
                " accuracy within 500 iterations." << endl;
    }
    return xp(p);
};
double Curvebase::y(double s)
{
    double p0 = 0, error = 1, toln = 1e-4, p, f;
    int iter = 0, iter_max = 500;
    while (error > toln && iter < iter_max)
    {
        f = integrate(pmin, p0, tol) - s * length;
        p = p0 - f / df(p0);
        error = fabs(p - p0);
        p0 = p;
        iter += 1;
    }
    if (iter == iter_max)
    {
        cout << "Solution not yet converged to desired"
                " accuracy within 500 iterations." << endl;
    }
    return yp(p);
};
//declaration of function
double Curvebase::I_1(double pmin, double pmax)
{
    //definition of function, integral using Simpson's rule
```

```
    double I1 = ((pmax - pmin) / 6) *
         (df(pmin) + 4 * df((pmin + pmax) / 2) + df(pmax));
    return I1;
}
//declaration of function
double Curvebase::I_2(double pmin, double pmax)
{
    //definition of variable, midpoint
    double gamma = (pmin + pmax) * 0.5;
    //definition of function, helping calculate error
    double I2 = I_1(pmin, gamma) + I_1(gamma, pmax);
    return I2;
}
//declaration of function, adaptive Simpson's Integration
double Curvebase::integrate(double pmin, double pmax, double tol)
{
    //compute I1 integral between values pmin and pmax
    double I1 = I_1(pmin, pmax);
    //compute I2 integral for estimating error
    double I2 = I_2(pmin, pmax);
    //calculate error estimation
    double errest = abs(I1 - I2);

    if (errest < 15 * tol) //decide if error is too big
    {
        //end loop if error estimate satisfies tolerance
        return I2;
    }
    else
    {
        //keep looping if error estimate does not satisfy
        //condition, algorithm will subdivide interval of
        //integration in two and apply ASI recursively
        return integrate(pmin, (pmin + pmax) / 2, tol / 2)
             + integrate((pmin + pmax) / 2, pmax, tol / 2);
    }
}
```

Appendix 3: line_v Header File

```
#ifndef linev
#define linev
#include "Curvebase.h"
class line_v : public Curvebase
{
    friend class Domain;
public:
    line_v(double pmin_in, double pmax_in, double d_in);
    line_v();
protected:
    double df(double p);
    double xp(double p);
    double yp(double p);
    double dxp(double p);
    double dyp(double p);
};
#endif
```

Appendix 4: line_v Source File

```
#include "Curvebase.h"
#include "line_v.h"
line_v::line_v() {};
line_v::line_v(double p_min_in, double p_max_in, double d_in)
{
    pmin = p_min_in;
    pmax = p_max_in;
    length = p_max_in - p_min_in;
    d = d_in;
};
double line_v::xp(double p)  {return d;};
double line_v::dxp(double p) {return 0.0;};
double line_v::yp(double p)  {return p;};
double line_v::dyp(double p) {return 1.0;};
double line_v::df(double p)   {return 1.0;};
```

Appendix 5: line_h Header File

```
#ifndef lineh
#define lineh
#include "Curvebase.h"
class line_h : public Curvebase
{
    friend class Domain;
public:
    line_h(double p_min_in, double p_max_in, double d_in);
    line_h();
protected:
    double df(double p);
    double xp(double p);
    double yp(double p);
    double dxp(double p);
    double dyp(double p);
};
#endif
```

Appendix 6: line_h Source File

```
#include "Curvebase.h"
#include "line_h.h"
line_h::line_h() {};
line_h::line_h(double p_min_in, double p_max_in, double d_in)
{
    pmin = p_min_in;
    pmax = p_max_in;
    length = p_max_in - p_min_in;
    d = d_in;
};
double line_h::xp(double p)  {return p;};
double line_h::dxp(double p) {return 1.0;};
double line_h::yp(double p)  {return d;};
double line_h::dyp(double p) {return 0.0;};
double line_h::df(double p)   {return 1.0;};
```

Appendix 7: lower_curve Header File

```
#ifndef lowercurve_h
#define lowercurve_h
#include "Curvebase.h"
class lower_curve : public Curvebase
{
    friend class Domain;
public:
    lower_curve(double p_min_in, double p_max_in, double d_in);
    lower_curve();
protected:
    double df(double p);
    double xp(double p);
    double yp(double p);
    double dxp(double p);
    double dyp(double p);
};
#endif
```

Appendix 8: lower_curve Source File

```
#include "Curvebase.h"
#include "lower_curve.h"
#include <cmath>
lower_curve::lower_curve() {};
lower_curve::lower_curve(double p_min_in, double p_max_in, double d_in)
{
    pmin = p_min_in;
    pmax = p_max_in;
    length = integrate(pmin, pmax, tol);
    d = d_in;
};
double lower_curve::xp(double p) {return p;};
double lower_curve::dxp(double p) {return 1;};
double lower_curve::yp(double p)
{
    if (p < -3) {
        return 0.5 * 1 / (1 + exp(-3 * (p + 6)));
    }
    else {
        return 0.5 * 1 / (1 + exp(3 * (p)));
    }
};
double lower_curve::dyp(double p)
{
    if (p < -3) {
        return 3.0 / 2 * exp(-3 * (p)-18.0) /
        ((1 + exp(-3 * (p)-18)) * (1 + exp(-3 * (p)-18)));
    }
    else {
        return -3.0 / 2 * exp(3 * (p)) /
        ((1 + exp(3 * (p))) * (1 + exp(3 * (p))));
    }
};
double lower_curve::df(double p)
{
    return sqrt(dyp(p) * dyp(p) + dxp(p) * dxp(p));
};
```

Appendix 9: main File

```
#include "lower_curve.h"
#include "line_v.h"
#include "line_h.h"
#include "Domain.h"
int main()
{
    // Task 2: Values for the boundary sides
    double pmin0 = -10.0, pmax0 = 5.0, d0 = 10.0;
    double pmin1 = 0.0, pmax1 = 3.0, d1 = 5.0;
    double pmin2 = -10.0, pmax2 = 5.0, d2 = 3.0;
    double pmin3 = 0.0, pmax3 = 3.0, d3 = -10.0;
    // Task 2: Creation of the boundary sides
    lower_curve curve_0 = lower_curve(pmin0, pmax0, d0);
    line_v line_v1 = line_v(pmin1, pmax1, d1);
    line_h line_h2 = line_h(pmin2, pmax2, d2);
    line_v line_v3 = line_v(pmin3, pmax3, d3);
    // Task 3: Creation of the domain and corresponding grid
    Domain D = Domain(curve_0, line_v1, line_h2, line_v3);
    D.grid_generation();
    // Task 4: Write the grid to a file
    D.writetofile();
    return 0;
}
```

Appendix 10: Domain Header File

```
#ifndef DOMAIN_H
#define DOMAIN_H
class Domain
{
public:
    Domain(lower_curve& curve_0, line_v& line_v1,
           line_h& line_h2, line_v& line_v3);
    // Task 3: Function generating grid
    void grid_generation();
    // Task 4: Function writing the grid to a file
    void writetofile();
private:
    // Grid intervals
    const static int n = 49, m = 19;
    // Interpolated grid points
    double x[n + 1][m + 1], y[n + 1][m + 1];
    // Grid points for boundary curve 0
    lower_curve curve_0;
    double* x_0, * y_0;
    // Grid points for boundary curve 1
    line_v line_v1;
    double* x_1, * y_1;
    // Grid points for boundary curve 2
    line_h line_h2;
    double* x_2, * y_2;
    // Grid points for boundary curve 3
    line_v line_v3;
    double* x_3, * y_3;
};
#endif
```

Appendix 11: Domain Source File

```
#include <iostream>
#include <fstream>
#include "lower_curve.h"
#include "line_v.h"
#include "line_h.h"
#include "Domain.h"
using namespace std;
Domain::Domain(lower_curve& curve_0_in, line_v& line_v1_in,
               line_h& line_h2_in, line_v& line_v3_in)
{
    // Allocating memory for:
    // Grid points for boundary curve 0
    curve_0 = lower_curve(curve_0_in);
    x_0 = new double[n + 1];
    y_0 = new double[n + 1];
    // Grid points for boundary curve 1
    line_v1 = line_v(line_v1_in);
    x_1 = new double[m + 1];
    y_1 = new double[m + 1];
    // Grid points for boundary curve 2
    line_h2 = line_h(line_h2_in);
    x_2 = new double[n + 1];
    y_2 = new double[n + 1];
    // Grid points for boundary curve 3
    line_v3 = line_v(line_v3_in);
    x_3 = new double[m + 1];
    y_3 = new double[m + 1];
};
void Domain::grid_generation()
{
    double h1 = 1.0 / n; //Stepsize x-direction
    double h2 = 1.0 / m; //Stepsize y-direction
    // Where boundary curves will have n+1 grid points
    for (int i = 0; i < n + 1; i++)
    {
    // Generating grid points:
    // Grid points for boundary curve 0
        x_0[i] = curve_0.x(i * h1);
        y_0[i] = curve_0.y(i * h1);
    // Grid points for boundary curve 2
        x_2[i] = line_h2.x(i * h1);
        y_2[i] = line_h2.y(i * h1);
    }
    // Where boundary curves will have m+1 grid points
    for (int i = 0; i < m + 1; i++)
    {
    // Generating grid points:
    // Grid points for boundary curve 1
        x_1[i] = line_v1.x(i * h2);
        y_1[i] = line_v1.y(i * h2);
    // Grid points for boundary curve 3
        x_3[i] = line_v3.x(i * h2);
        y_3[i] = line_v3.y(i * h2);
    }
    int gridpoints_tot = 0;
```

```
    // Interpolating interior grid points using
    // The algebraic grid generation formula
    for (int i = 0; i < n + 1; i++)
    {
        for (int j = 0; j < m + 1; j++)
        {
           x[i][j] = (1 - i * 1.0 / n) * x_3[j] + i * 1.0 / n * x_1[j]
                   + (1 - j * 1.0 / m) * x_0[i] + j * 1.0 / m * x_2[i]
                   - (1 - i * 1.0 / n) * (1 - j * 1.0 / m) * (-10)
                   - i * 1.0 / n * (1 - j * 1.0 / m) * (5)
                   - (1 - i * 1.0 / n) * j * 1.0 / m * (-10)
                   - i * 1.0 / n * j * 1.0 / m * (5);
           y[i][j] = (1 - i * 1.0 / n) * y_3[j] + i * 1.0 / n * y_1[j]
                   + (1 - j * 1.0 / m) * y_0[i] + j * 1.0 / m * y_2[i]
                   - (1 - i * 1.0 / n) * (1 - j * 1.0 / m) * (0)
                   - i * 1.0 / n * (1 - j * 1.0 / m) * (0)
                   - (1 - i * 1.0 / n) * j * 1.0 / m * (3)
                   - i * 1.0 / n * j * 1.0 / m * (3);
           gridpoints_tot += 1;
        }
    }
    cout << "The total amount of gridpoints is: " << gridpoints_tot << endl;
};
void Domain::writetofile() {     // TASK 4
    ofstream boundary_h;
    boundary_h.open("boundary_h.txt");
    for (int i = 0; i < n + 1; i++)
    {
        // Store boundary grid points in file:
        // Grid points for boundary curve 2
        boundary_h << x_2[i];
        boundary_h << "\t";
        boundary_h << y_2[i];
        boundary_h << "\t";
        // Grid points for boundary curve 0
        boundary_h << x_0[i];
        boundary_h << "\t";
        boundary_h << y_0[i];
        boundary_h << "\n";
    }
    boundary_h.close();
    ofstream boundary_v;
    boundary_v.open("boundary_v.txt");
    for (int i = 0; i < m + 1; i++) {
        // Store boundary grid points in file:
        // Grid points for boundary curve 3
        boundary_v << x_3[i];
        boundary_v << "\t";
        boundary_v << y_3[i];
        boundary_v << "\t";
        // Grid points for boundary curve 1
        boundary_v << x_1[i];
        boundary_v << "\t";
        boundary_v << y_1[i];
        boundary_v << "\n";
    }
    boundary_v.close();
```

```
    ofstream interior_x;
    interior_x.open("interior_x.txt");
    for (int i = 0; i < n + 1; i++)
    {
        for (int j = 0; j < m + 1; j++)
        {
        // Store x - coordinates for interpolated
        // interior grid points in file:
            interior_x << x[i][j];
            interior_x << "\t";
        }
        interior_x << "\n";
    }
    interior_x.close();
    ofstream interior_y;
    interior_y.open("interior_y.txt");
    for (int i = 0; i < n + 1; i++)
    {
        for (int j = 0; j < m + 1; j++)
        {
        // Store y - coordinates for interpolated
        // interior grid points in file:
            interior_y << y[i][j];
            interior_y << "\t";
        }
        interior_y << "\n";
    }
    interior_y.close();
};
```

Appendix 12: Matlab File

```
clear all; close all; clc

% Load boundary files
load boundary_h.txt -ascii
load boundary_v.txt -ascii

% Load interior files
load interior_x.txt -ascii
load interior_y.txt -ascii

% Creating vertical grid lines
for i = 1:50
   plot(interior_x(i,:),interior_y(i,:),'m')
   hold on
end

% Creating horizontal grid lines
for i = 1:20
   plot(interior_x(:,i),interior_y(:,i),'m')
end

% Creating grid points
for i = 1:20
    for j = 1:50
        plot(interior_x(j,i), interior_y(j,i),'k.')
    end
end

% Marking boundary grid points with stars
plot(boundary_v(:,1), boundary_v(:,2),'k*',
boundary_v(:,3), boundary_v(:,4),'k*')
plot(boundary_h(:,1), boundary_h(:,2),'k*',
boundary_h(:,3), boundary_h(:,4),'k*')
axis([-11 6 -1 4]);
```