Program construction in C++ for Scientific Computing - Project 4

Pia Callmer, Jonas Fey

December 2022

Task 1

The class `Domain` is redesigned using smart pointers and vectors. Smart pointers are mainly used for the curvebases, to ensure that if one curvebase is deleted from the outside, the grid will still have the curvebase for creating the grid. In addition, the x and y C-style arrays containing the x and y coordinates are now vectors. In that way it is easier to resize the shape of the array instead of deleting it and creating a new one. For example when generating a grid, we can resize it easily by using `.resize(shape)`.

Task 2

After implementing the standard operations, such as addition and multiplication by a scalar, the differential operators $\partial/\partial x, \partial/\partial y$ and Δ are implemented as well. To ensure second order accuracy a smooth mapping $\phi : [0,1]^2 \longrightarrow \Omega$ is needed. The reference domain $[0,1]^2$ is a $n \times m$ uniform grid as described in project 3. The derivatives can be calculated using the chain rule of differentiation, which then leads to

$$\begin{aligned}
\frac{\partial u}{\partial x} &= \frac{1}{\det J}\left(\frac{\partial u}{\partial \xi}\frac{\partial \Phi_y}{\partial \eta} - \frac{\partial u}{\partial \eta}\frac{\partial \Phi_y}{\partial \xi}\right) \\
\frac{\partial u}{\partial y} &= \frac{1}{\det J}\left(\frac{\partial u}{\partial \eta}\frac{\partial \Phi_x}{\partial \xi} - \frac{\partial u}{\partial \xi}\frac{\partial \Phi_x}{\partial \eta}\right) \\
\text{with} \quad J &= \begin{bmatrix} \frac{\partial \Phi_x}{\partial \xi} & \frac{\partial \Phi_y}{\partial \xi} \\ \frac{\partial \Phi_x}{\partial \eta} & \frac{\partial \Phi_y}{\partial \eta} \end{bmatrix}
\end{aligned}$$

Each of these partial derivatives, e.g. $\frac{\partial \Phi_x}{\partial \xi}$, are approximated using the central difference quotient, which makes the approximation second order accurate. The exceptions are at the boundaries, where we use forward - or backward differences respectively, depending on, which grid points are available. For example we would use forward differences on the left side and backwards differences on the right side of the function. This makes the approximations at the boundaries only first order accurate.
Approximating the Laplace operator $\Delta f(x,y) = \partial_x \partial_x f + \partial_y \partial_y f$ can be done by using the partial derivatives $\frac{\partial u}{\partial x}$ and $\frac{\partial u}{\partial y}$ twice respectively and summing up the solutions. Implementing this concept is a lot easier than using the chain rule of differentiation's on $\frac{\partial u}{\partial x}$ or $\frac{\partial u}{\partial y}$.

Task 3

In this task the `GFkt` class is used to discretize the function

$$u(x,y) = \sin\left(\frac{x^2}{100}\right)\cos\left(\frac{x}{10}\right) + y, \tag{1}$$

which is visualized in Figure 1. For this task the same grid as in project 3 is used with 50×20 grid points. Comparing the discretized functions and derivatives to the analytic solution as shown in Figure 2, 3 and 5, one can see that the result is very similar. However, the error plots in Figure 4a show that the error for $\frac{\partial u}{\partial x}$

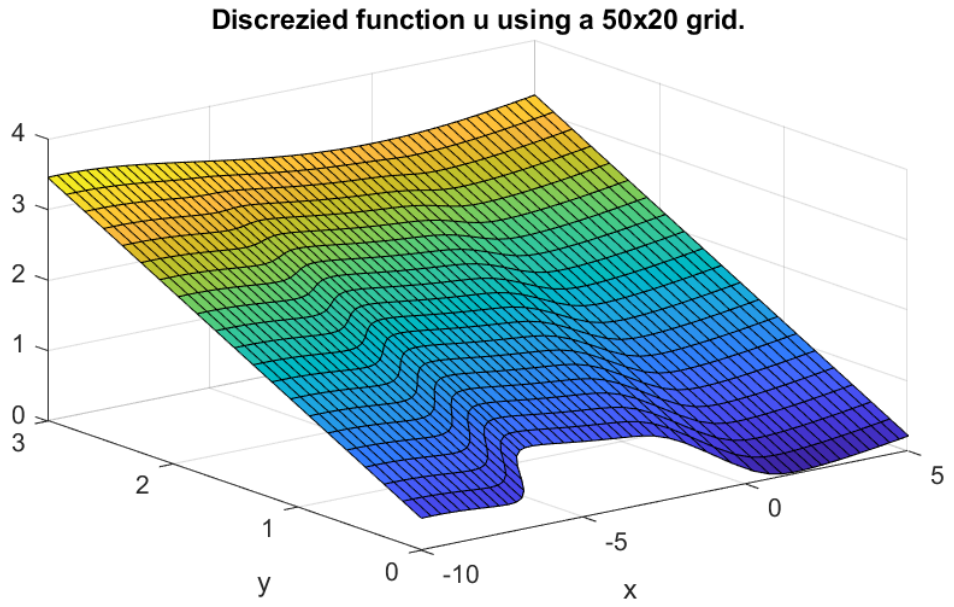
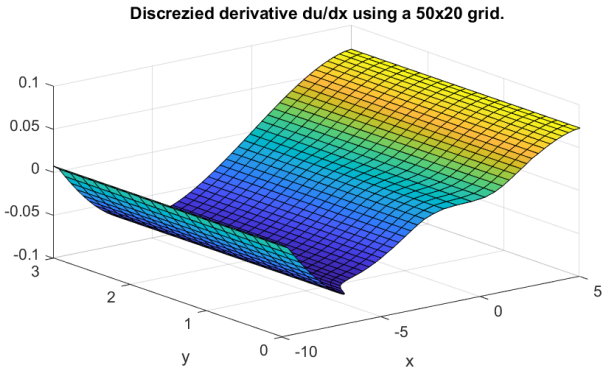


Figure 1: Function u on a 50×20 grid for $x \in [-10, 5]$ and $y \in [0, 3]$.

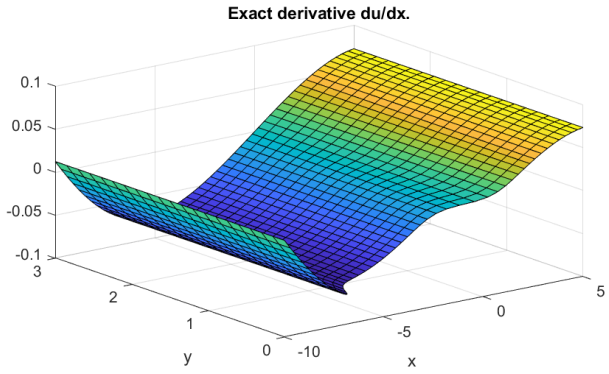
is a lot higher at the boundaries. The same property can be seen for the error of the Laplace function in figure 5c. The error of the approximation $\frac{\partial u}{\partial y}$ is not higher at the boundaries, but rather a bit higher where the grid points are not equally spaced. But it is important to mention that the error is a lot smaller, since its in the range of 10^{-5} compared to the maximum error of $\frac{\partial u}{\partial x}$ with magnitude 10^{-3} at the boundaries.

Especially at the left boundary, where $x = -10$, the error of $\frac{\partial u}{\partial x}$ is a lot higher than on the boundary at $x = 5$. This is due to the nature of the grid function (1) and the chosen domain of $x = [-10, 5]$. Looking at the function $u(x, y = 0)$ in Figure 6 one can see that the function at the boundary values, i.e. at $x = -10$ is almost at its maximum value, while the slope at $x = 5$ is quite steep and almost linear. This leads to the difference in the approximation errors at the boundaries, since at $x = -10$ the gradient of the function changes a lot quicker than at $x = 5$. Therefore, more grid points are necessary to achieve the same level of accuracy. If the range of x would be $x = [-10, 10]$ instead, such that $u(x, y = 0)$ is symmetric around the axis, the error would be symmetric too, which is depicted in Figure 6b.

The Laplacian has a larger error than the partial derivatives, since it is composed by the functions of the partial derivatives. Because the Laplacian uses finite differences on the partial derivatives $\frac{\partial u}{\partial x}$ and $\frac{\partial u}{\partial y}$, two points to either side of the point have to be evaluated to approximate the Laplacian. This then leads to the case that not only the first but also the second element from the boundary suffers from the less accurate solution that using the forward (or backward) finite difference brings with it.

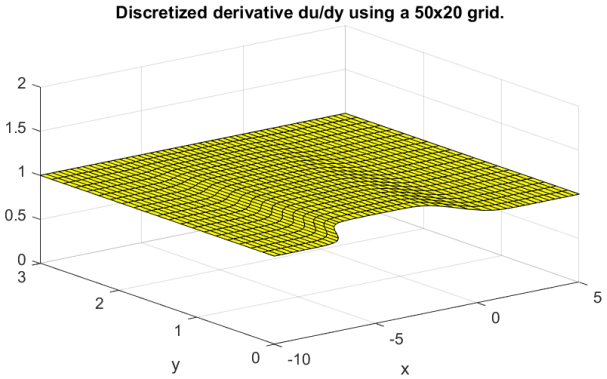


(a) Discretized derivative $\frac{\partial u}{\partial x}$ for a 50×20 grid.

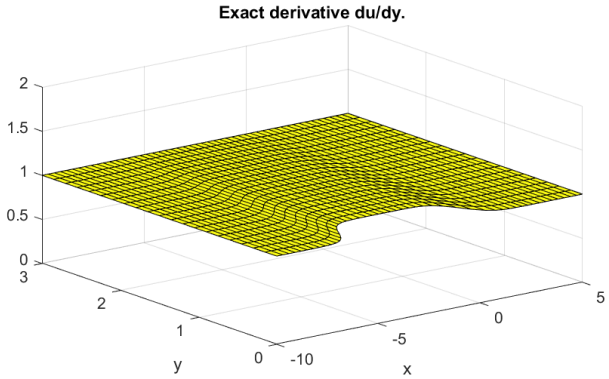


(b) Analytical derivative $\frac{\partial u}{\partial x}$.

Figure 2: Partial derivative $\frac{\partial u}{\partial x}$ for $x \in [-10, 5]$ and $y \in [0, 3]$.

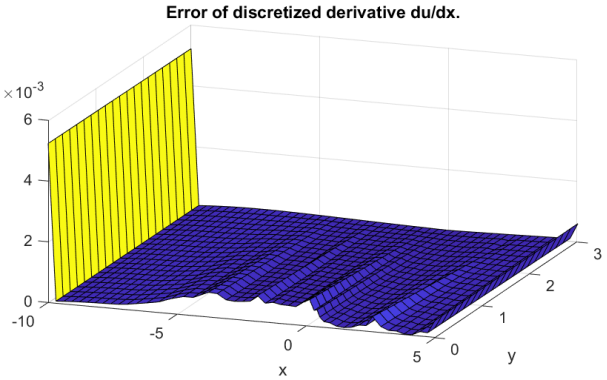


(a) Discretized derivative $\frac{\partial u}{\partial y}$ for a 50×20 grid.

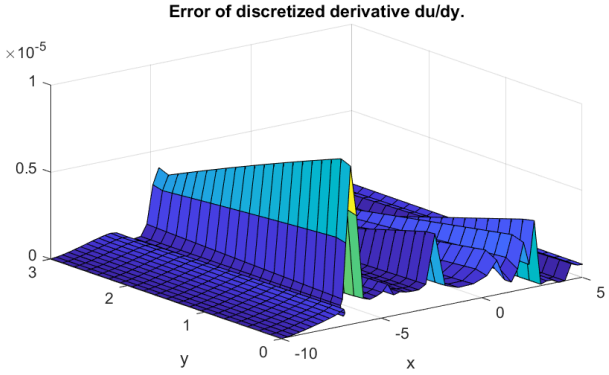


(b) Analytical derivative $\frac{\partial u}{\partial y}$.

Figure 3: Partial derivative $\frac{\partial u}{\partial y}$ for $x \in [-10, 5]$ and $y \in [0, 3]$.

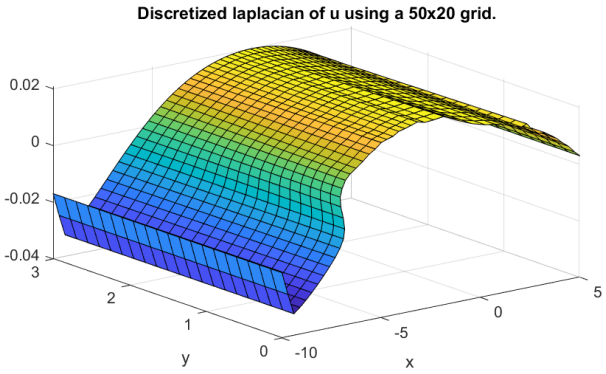


(a) Errors of discretized derivative $\frac{\partial u}{\partial x}$.

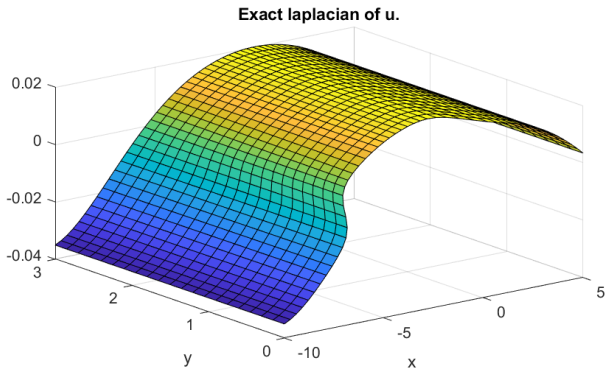


(b) Errors of discretized derivative $\frac{\partial u}{\partial y}$.

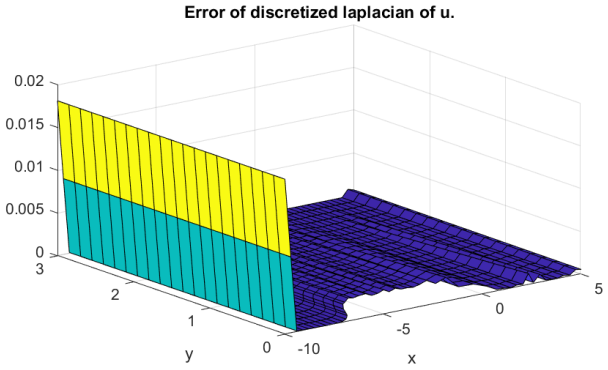
Figure 4: Errors of discretized derivatives on a 50×20 grid for $x \in [-10, 5]$ and $y \in [0, 3]$.



(a) Discretized laplacian Δu for a 50×20 grid.

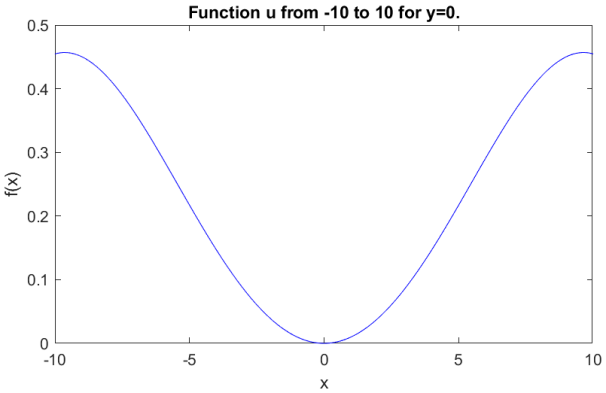


(b) Analytical laplacian Δu.

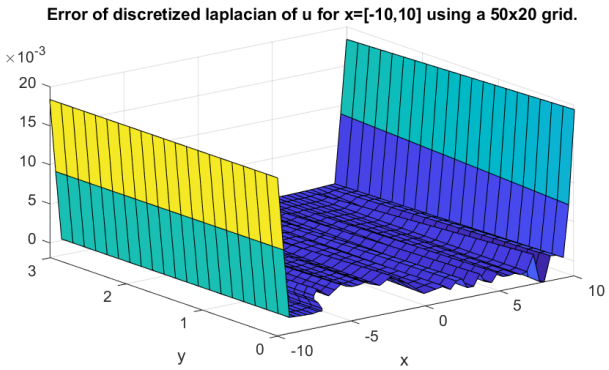


(c) Error of discretized laplacian Δu.

Figure 5: Laplacian Δu for a 50×20 grid for $x \in [-10, 5]$ and $y \in [0, 3]$.



(a) Function $u(x, y = 0)$ for $x \in [-10, 10]$.



(b) Error of discretized laplacian Δu for a 50×20 grid.

Figure 6: Function $u(x, y = 0)$ and Δu for $x \in [-10, 10]$ and $y \in [0, 3]$

A Appendix – Code files

A.1 main.cpp

```
#include <iostream>
#include <memory>
#include <cmath>
#include "curvebase.hpp"
#include "xvertical.hpp"
#include "yhorizontal.hpp"
#include "nonconstcurve.hpp"
#include "domain.hpp"
#include "GFkt.hpp"
#include "matrix.hpp"
#include<cstdio>

using namespace std;

// Function u(x,y) as described in the exercise
double func(double x, double y){
    return sin((x*x)/100.0) * cos(x/10.0) + y;

}



int main()
{

    // Defining the domain as described by the exercise
    shared_ptr<Nonconstcurve> s1 = make_shared<Nonconstcurve>(-10.0, 5.0);
    shared_ptr<Xvertical> s2 = make_shared<Xvertical>(0.0, 3.0, 5.0);
    shared_ptr<Yhorizontal> s3 = make_shared<Yhorizontal>(5.0, -10.0, 3.0)
       ↪ ;
    shared_ptr<Xvertical> s4 = make_shared<Xvertical>(3.0, 0.0, -10.0);


    // Generate domain
    shared_ptr<Domain> mydomain = make_shared<Domain>(s1,s2,s3,s4);

    // Choose size of the grid to be generated
    int N = 50;
    int M = 20;
    mydomain->generate_grid(N,M);

    // Define grid functions, all on the same domain to allow for the
       ↪ operations to work
    GFkt gridfun(mydomain);
    GFkt gridfundx(mydomain);
    GFkt gridfundy(mydomain);


    // Make a matrix and fill it with the function values of u for the
       ↪ given grid points
```

```
Matrix m1(N+1,M+1);

double v1[(N+1)*(M+1)];
for(int j=0; j< (M+1); j++){
    for(int i=0; i< (N+1); i++){
        v1[j*(N+1)+i] = func(mydomain->X_()[j*(N+1)+i], mydomain->Y_()
           ↪ [j*(N+1)+i]);
    }
}
m1.fillMatrix(v1, (N+1)*(M+1));
gridfun.setFunction(m1);

cout << "Gridfunction for u generated!" << endl;

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

// Calculate the partial derivatives and store them as Gridfunctions
   ↪ gridfundx and gridfundy
gridfun.DOxy(&gridfundx, &gridfundy);

// Get the laplacian of u and store it as a Gridfunction
   ↪ gridfunlaplace
GFkt gridfunlaplace = gridfun.Laplacian();


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);

cout << "Saved grid to <grid_out.bin>!" << endl;


// Save discretized u function to the file 'u_out.bin'
counter = 0;
len = (mydomain->M_()+1)*(mydomain->N_()+1);
double gridu[len];
for(int i=0; i<mydomain->N_()+1;i++){
    for(int j=0; j<mydomain->M_()+1; j++){
        gridu[j*(mydomain->N_()+1)+i] = gridfun.U().Matx()[i][j];
    }
```

```
}

FILE *fpu;
fpu =fopen("u_out.bin","wb");
fwrite(gridu,sizeof(double),len,fpu);
fclose(fpu);
cout << "Saved u to <u_out.bin>!" << endl;


// Save discretized derivative du / dx
counter = 0;
len = (mydomain->M_()+1)*(mydomain->N_()+1);
double griddux[len];
for(int i=0; i<mydomain->N_()+1;i++){
    for(int j=0; j<mydomain->M_()+1; j++){
        griddux[j*(mydomain->N_()+1)+i] = gridfundx.U().Matx()[i][j];
    }
}

FILE *fpdux;
fpdux =fopen("dux_out.bin","wb");
fwrite(griddux,sizeof(double),len,fpdux);
fclose(fpdux);
cout << "Saved du / dx to <dux_out.bin>!" << endl;



// Save discretized derivative du / dy
counter = 0;
len = (mydomain->M_()+1)*(mydomain->N_()+1);
double gridduy[len];
for(int i=0; i<mydomain->N_()+1;i++){
    for(int j=0; j<mydomain->M_()+1; j++){
        gridduy[j*(mydomain->N_()+1)+i] = gridfundy.U().Matx()[i][j];
    }
}

FILE *fpduy;
fpduy =fopen("duy_out.bin","wb");
fwrite(gridduy,sizeof(double),len,fpduy);
fclose(fpduy);
cout << "Saved du / dy to <duy_out.bin>!" << endl;


// Save discretized Laplacian nabla u
counter = 0;
len = (mydomain->M_()+1)*(mydomain->N_()+1);
double gridlaplace[len];
for(int i=0; i<mydomain->N_()+1;i++){
    for(int j=0; j<mydomain->M_()+1; j++){
        gridlaplace[j*(mydomain->N_()+1)+i] = gridfunlaplace.U().Matx
            ↪ ()[i][j];
    }
}
```

```
    FILE *fplaplace;
    fplaplace =fopen("dlaplace_out.bin","wb");
    fwrite(gridlaplace,sizeof(double),len,fplaplace);
    fclose(fplaplace);
    cout << "Saved nabla u to <dlaplace_out.bin>!" << endl;



    // 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 << "Saved curvebases to <curve_out.bin>!" << endl;

    return 0;
}
```

A.2 GFkt.hpp

```
#include "domain.hpp"
#include "matrix.hpp"

#ifndef GFKT.HPP
#define GFKT.HPP

class GFkt {

private:
    Matrix u; // Stores the function values at the grid points defined by
       ↪ grid
    shared_ptr<Domain> grid; // Object of class Grid. Represents a grid
       ↪ consisting of four curvebases


public:
    // Constructor
    GFkt(shared_ptr<Domain> grid_) : u(grid_->N_()+1,grid_->M_()+1), grid(
       ↪ grid_) {}

    // Copy constructor
    GFkt(const GFkt& U) : u(U.u), grid(U.grid) {}

    // Operator overload
    // Addition by another Gridfunction
    GFkt operator+(const GFkt& U) const;

    // Multiplication with a scalar
    GFkt operator*(const double p) const;

    // Generate partial derivatives of u and save them into dx and dy
    void D0xy(GFkt *dx, GFkt *dy) const;

    // Generate Laplacian of u and return it as a Gridfunction
    GFkt Laplacian() const;

    // Getter functions
    Matrix U() const {return u;};
    shared_ptr<Domain> Grid() const {return grid;};

    // Set function values for u
    void setFunction(Matrix u_new);

    // Setter for a new grid
    void setGrid(shared_ptr<Domain> newgrid);

    // Print function
    void printGFkt() const;

};

#endif // GFKT
```

A.3 GFkt.cpp

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

using namespace std;

// Operation overload, addition by another grid function
GFkt GFkt::operator+(const GFkt& U) const {
    if(grid == U.grid){ // Defined on the same Grid
        GFkt tmp(*this);
        tmp.u += U.u; // Use Matrix += operator which is overloaded
        return tmp;
    } else {
        cout << "Grids are not identical, + operation failed!" << endl;
        exit(0);
    }
}


// Operation overload, multiplication with a scalar
GFkt GFkt::operator*(const double p) const {
    GFkt tmp(*this);
    tmp.u *= p; // Using the overloaded *= operator of class Matrix
    return tmp;
}


// Calculate the derivatives of u and store them in Gridfunctions dx and
   ↪ dy
void GFkt::D0xy(GFkt *dx, GFkt *dy) const {

    // If the grid does not match, set it to the one of u
    if(grid != dx->Grid()){
        dx->setGrid(grid);
    }

    if(grid != dy->Grid()){
        dy->setGrid(grid);
    }

    // Size of grid
    int m = grid->M_();
    int n = grid->N_();

    // Create matrices that will contain the partial derivatives
    Matrix du_dx(n+1, m+1);
    Matrix du_dy(n+1, m+1);

    // Step size on the reference grid
    double h1 = 1.0/n;
    double h2 = 1.0/m;
```

```
// Vectors to fill the derivative matrices
double dux[(n+1)*(m+1)];
double duy[(n+1)*(m+1)];

// Iterate over every grid point
for(int j = 0; j <= m; ++j){
    for(int i = 0; i <= n; ++i){

        // Define indices, this is done because the finite differences
           ↪  on the boundaries us forward/backward differences
        // instead of central differences.
        double lix = i-1; // Left index for xi variable
        double rix = i+1; // Right index for xi variable
        double liy = -(n+1); // Left index for eta variable
        double riy = n+1; // Right index for eta variable
        double hxi = 2*h1; // Standard step size (the denominator of
           ↪ the finite difference) in xi direction
        double heta = 2*h2; // Standard step size (the denominator of
           ↪ the finite difference) in eta direction

        if (i==0){ // if i==0, we are at the left boundary in xi
           ↪ direction
            lix = i; // Therefore the left index is adjusted (use
               ↪ forward difference)
            hxi = h1; // Step size is also adjusted
        } else if( i== n){ // if i ==n we are at the right boundary
            rix = i ; // Adjust parameters for backward difference
            hxi = h1;
        }

         if (j==0){ // The same as for i but now in eta direction
            liy = 0; // Use forward difference
            heta = h2;
        } else if( j== m){
            riy = 0; // Use backward difference
            heta = h2;
        }

        // Define required parameters for the equations to solve for
           ↪ du/dx and du/dy as presented in the lecture
        double dphix_dxi;
        double dphix_deta;
        double dphiy_dxi;
        double dphiy_deta;

        // Using finite differences to calculate  the partial
           ↪ derivatives of Phi. These values are already stored in
           ↪ the physical
        // coordinates grid->X_() and grid->Y_()
        dphix_dxi = (grid->X_()[j*(n+1)+rix] - grid->X_()[j*(n+1)+lix
           ↪ ]) / hxi;

        dphix_deta = (grid->X_()[j*(n+1)+i+riy] - grid->X_()[j*(n+1)+i
           ↪ + liy]) / heta;
```

```
            dphiy_dxi = (grid->Y_()[j*(n+1)+rix] - grid->Y_()[j*(n+1)+lix
               ↪ ]) / hxi;

            dphiy_deta = (grid->Y_()[j*(n+1)+i+riy] - grid->Y_()[j*(n+1)+i
               ↪ +liy]) / heta;


            // Calculate the determinant as described in the lecture
            double DetJ = dphix_dxi * dphiy_deta - dphix_deta * dphiy_dxi;

            // Define parameters for the partial derivatives du/dxi and du
               ↪ /deta
            double du_dxi;
            double du_deta;

            // Again using finite differences:
            if (i==0){ // If i == 0, use forward difference
                du_dxi = (-u.Matx()[i][j] + u.Matx()[i+1][j]) / h1;
            } else if (i==n){ // If i == n use backward difference
                du_dxi = (-u.Matx()[i-1][j] + u.Matx()[i][j]) / h1;
            } else { // Everywhere else use central difference
                du_dxi = (-u.Matx()[i-1][j] + u.Matx()[i+1][j]) / (2*h1);
            }

            if (j==0){ // if j == 0 use forward difference
                du_deta = (-u.Matx()[i][j] + u.Matx()[i][j+1]) / h2;
            } else if(j==m){ // if j == m use backward difference
                du_deta = (-u.Matx()[i][j-1] + u.Matx()[i][j]) / h2;
            } else { // Everywhere else use central difference
                du_deta = (-u.Matx()[i][j-1] + u.Matx()[i][j+1]) / (2*h2);
            }

            // Now calcualte the partial derivatives at the current grid
               ↪ point as defined by the equation in the lecture
            dux[j*(n+1)+i] = 1 / DetJ * (du_dxi * dphiy_deta - du_deta *
               ↪ dphiy_dxi);
            duy[j*(n+1)+i] = 1 / DetJ * (du_deta * dphix_dxi - du_dxi *
               ↪ dphix_deta);
        }
    }

    // Fill the matrices with the vectors that contain the derivative
       ↪ information
    du_dx.fillMatrix(dux, (m+1)*(n+1));
    du_dy.fillMatrix(duy, (m+1)*(n+1));

    // Set derivative functions for the grid functions
    dx->setFunction(du_dx);
    dy->setFunction(du_dy);

}

// Function that returns the laplacian of u as a gridfunction
```

```
GFkt GFkt::Laplacian() const {

    // Make copies of the grid to store the first derivatives
    GFkt ux = GFkt(*this);
    GFkt uy = GFkt(*this);

    // Make copies of the grid to store the second derivatives
    GFkt uxx = GFkt(*this);
    GFkt uyy = GFkt(*this);
    GFkt uxy = GFkt(*this);

    // Calculate the first derivatives
    D0xy(&ux, &uy);

    // Calculate the second derivatives
    ux.D0xy(&uxx, &uxy);
    uy.D0xy(&uxy, &uyy);

    // The laplacian is the sum of uxx and uyy
    uxx.u += uyy.u;

    return uxx;
}

// Function to set a function on the grid
void GFkt::setFunction(Matrix u_new){
    u = u_new;
    cout << "Created new function on the grid!" << endl;
}

// Function to set a new grid
void GFkt::setGrid(shared_ptr<Domain> newgrid){
    grid = newgrid;
}

// Print function
void GFkt::printGFkt() const {
    grid->printGrid();
    u.printMatrix();
}
```

A.4 Domain.hpp

```
#include <memory>
#include <vector>
#include "curvebase.hpp"
#ifndef DOMAIN_HPP

#define DOMAIN_HPP

using namespace std;

class Domain{

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


    // Vectors that will store the coordinates of the grid points
    vector<double> x_;
    vector<double> y_;


    // Size of the grid
    int n_ = 0;
    int m_ = 0;

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


public:
    // Constructor
    Domain(shared_ptr<Curvebase> s1, shared_ptr<Curvebase> s2, shared_ptr<
       ↪ Curvebase> s3, shared_ptr<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();

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

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

    vector<double> X_() const {return x_;};
    vector<double> Y_() const {return y_;};

    int N_() const {return n_;};
```

```
    int M_() const {return m_;};
};

#endif
```

A.5 Domain.cpp

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

using namespace std;

// Constructor
Domain::Domain(shared_ptr<Curvebase> s1, shared_ptr<Curvebase> s2,
    ↪ shared_ptr<Curvebase> s3, shared_ptr<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(){
    x_.clear();
    y_.clear();
}

// Consistency check
bool Domain::check_consistency(){
    for(int i = 0; i < 4; i++){
        shared_ptr<Curvebase> current = sides[i];
        shared_ptr<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 an old grid already exists, change the sizes of the vectors and
       ↪ overwrite its data with the new grid
    if(m < 1 || n < 1) exit(1);
    if(m_ != m && n_ != n){
        x_.resize((m+1) * (n+1));
        y_.resize((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.6 Curvebase.hpp

```
#ifndef CURVEBASE_HPP
#define CURVEBASE_HPP


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

    virtual 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.7 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.8 Matrix.hpp

```
#ifndef MATRIX_HPP
#define MATRIX_HPP


// Header file for Matrix object
class Matrix {
private:
    int rows;
    int cols;
    double **matx;
public:
    //Constructors
    Matrix(int m); // Creates mxm Matrix with all entries 0
    Matrix(int n, int m); // Creates nxm Matrix with all entries 0
    Matrix(int m, double* a, int arraySize); // Creates mxm Matrix with
       ↪ values given by vector a with size arraySize
    Matrix(const Matrix&); // Copy constructor

    // Destructor
    ~Matrix();

    //Operation overloading
    Matrix& operator=(const Matrix&);
    Matrix& operator+=(const Matrix&);
    Matrix& operator*=(const Matrix&);
    Matrix& operator*=(const double);

    //Getter (member) functions
    int Rows() const {return rows;};
    int Cols() const {return cols;};
    double** Matx() const {return matx;};

    //Member functions
    double p2norm() const;
    double maxnorm() const;
    void printMatrix() const;
    void fillMatrix(double *, int);
    void fillNumber(double );
    void fillDiagonal(double);
};
#endif
```

A.9 Matrix.cpp

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

#include "matrix.hpp"
using namespace std;


//Constructor for a square matrix with dimensions m x m
Matrix::Matrix(int m){
    rows = m;
    cols = m;

    // Allocate new Memory
    matx = new double*[rows];
    for(int i = 0; i < rows; i++){
        matx[i] = new double[cols];
        for(int j = 0 ; j < cols ; j++){
            matx[i][j] = 0; // Assume 0 Matrix as initial value
        }
    }

}

Matrix::Matrix(int n, int m){
    rows = n;
    cols = m;

    // Allocate new Memory
    matx = new double*[rows];
    for(int i = 0; i < rows; i++){
        matx[i] = new double[cols];
        for(int j = 0 ; j < cols ; j++){
            matx[i][j] = 0; // Assume 0 Matrix as initial value
        }
    }

}

// Constructor for a square matrix with dimensions m x m and values given
   ↪ by vector a
Matrix::Matrix(int m, double* a, int arraySize){
    if(m*m == arraySize){
        rows = m;
        cols = m;

        // Fill Matrix with a
        matx = new double*[rows];
        for(int i = 0; i < rows; i++){
            matx[i] = new double[cols];
            for(int j = 0 ; j < cols ; j++){
                matx[i][j]= a[j*cols+i];
            }
```

```
        }
    } else cout << "ERROR Matrix Constructor: Dimension of a does not fit
       ↪ into Matrix with size: " << m << "x" << m << endl;
}


// Copy constructor, construct similar matrix with same attributes as
   ↪ given matrix M
Matrix::Matrix(const Matrix& M){

    //take over the attributes
    rows = M.Rows();
    cols = M.Cols();

    //build new matrix
    matx = new double*[rows];

    // loop over each element
    for (int i = 0 ; i< rows ; i++){
        // allocate the memory
        matx[i] = new double[cols];
        for(int j = 0 ; j < cols ; j++)
            matx[i][j] = M.Matx()[i][j]; // Copy values
    }
}


// Destructor for the matrix
Matrix::~Matrix() {
    for (int i = 0; i < rows; i++) {
        delete[] matx[i];
    }
        delete[] matx;
}


//Overrule copy operator
Matrix& Matrix::operator=(const Matrix& M){
    if (this != &M){
        // Create new Matrix with the number of rows and coloums being the
           ↪  same as M
        rows = M.Rows();
        cols = M.Cols();
        matx = new double*[rows];

        // Filling the values
        for (int i = 0 ; i< rows ; i++){
            matx[i] = new double[cols];

            for(int j = 0 ; j < cols ; j++)
                // copy values
                matx[i][j] = M.Matx()[i][j];
            }
        }
```

```
    return *this; // dereferencing
}

//Overload operator +=
Matrix& Matrix::operator+=(const Matrix& M ){
    if (rows == M.Rows() && cols == M.Cols()){

         // loop over each element
        for (int i = 0 ; i< rows ; i++){
            for(int j = 0 ; j < cols ; j++)
                matx[i][j] += M.Matx()[i][j]; // addition element-wise
        }
    } else
        cout << "␣ERROR␣+=␣:␣The␣matrix␣shapes␣do␣not␣match␣for␣addition!␣
           ↪ Left␣Matrix␣" << rows  << "␣x␣" << cols << "␣Right␣matrix␣"
           ↪ << M.Rows() <<"␣x␣"<< M.Cols() << endl;
    return *this;

}

//Overload operator *= for Matrix multiplications
Matrix& Matrix::operator*=(const Matrix& M ){
    if (cols == M.Rows()){

        double ** lmatx = new double*[rows];

        // Copy Matrix
        lmatx = matx;

        // Create resulting matrix of size: rows x M.Cols()
        matx = new double*[rows];
        for(int i = 0 ; i < M.Cols() ; i++){
            matx[i] = new double[M.Cols()];
        }

         // loop over each element
        for (int i= 0; i< rows; i++ ){
            for (int j = 0 ; j< M.Cols() ; j++){
                double sum = 0;

                //get the linear combination to compute element i,j
                for (int k=0; k< rows; k++)
                    sum += lmatx[i][k]*M.Matx()[k][j];
                matx[i][j] = sum;
            }
        }
        //Set the new amount of columns (rows stay the same)
        cols = M.Cols();
    }
    else
        cout << "␣ERROR␣*=:␣The␣matrix␣shapes␣do␣not␣match␣for␣
           ↪ multiplication!␣Left␣Matrix␣" << rows  << "␣x␣" << cols << "
           ↪ ␣Right␣matrix␣" << M.Rows() <<"␣x␣"<< M.Cols() << endl;
    return *this;
```

```
}

// Overload operator *= To multiply matrix with scalar value
Matrix& Matrix::operator*=(const double a ){

         // loop over each element
        for (int i = 0 ; i< rows ; i++){
            for(int j = 0 ; j < cols ; j++)
                matx[i][j] *= a; // multiply element-wise
        }
    return *this;
}


// Member function to print matrix:
void Matrix::printMatrix() const{
  for (int i = 0 ; i< rows ; i++){
    cout << "|";
    for(int j = 0 ; j < cols-1 ; j++)
        cout << matx[i][j] << ", " ;
    cout << matx[i][cols-1] << "|" << endl ;
    }
}

// Implementing the 2 norm, i.e. sum over all elements, take the square
   ↪ element-wise and take the root of the sum
double Matrix::p2norm() const{
    double res = 0;
     for (int i = 0 ; i< rows ; i++){
            for(int j = 0 ; j < cols ; j++)
                // Add square of each element
                res += matx[i][j]*matx[i][j];
    }
    // Take the root
    res = sqrt(res);
    return res;
}

//Maximum norm returns the absolute largest value of the matrix
double Matrix::maxnorm() const{
    // Initialize max value
    double current_max = -1.0;
    for (int i = 0 ; i< rows ; i++){
        for(int j = 0 ; j < cols ; j++){
            // Update if absolute value is larger
            if (abs(matx[i][j]) > current_max){
                current_max = abs(matx[i][j]);
            }
        }
    }
    return current_max;
}
```

```
// Function which fills all elements with the vector a of length n*m
   ↪ column-wise
void Matrix::fillMatrix(double *a, int arraySize){

    // Check if dimension align
    if ( rows*cols == arraySize ) {
        for (int i= 0 ; i< rows ; i++){
            for(int j = 0 ; j < cols ; j++) {
                // Fill value with value given by the vector
                matx[i][j]= a[j*rows+i];
            }
        }
    } else

     cout << "ERROR␣FILLMATRIX:␣The␣length␣of␣the␣array␣does␣not␣match␣the
        ↪ ␣matrix␣shape:␣" << rows << "␣x␣" << cols << "␣Got␣shape:␣" <<
        ↪ a << "␣Expected:␣" << rows*cols  << endl;
}


// Function which fills all elements of the matrix with the same double a
void Matrix::fillNumber(double a){
    for (int i= 0 ; i< rows ; i++){
        for(int j = 0 ; j < cols ; j++)
            matx[i][j]= a;
    }
}

// Fill diagonal of matrix with value a
void Matrix::fillDiagonal(double a){
    int smallersize;
    // if non-square matrix find min(rows, cols)
    if(rows < cols){
        smallersize = rows;
    }
    else {
        smallersize = cols;
    }

    for (int i= 0 ; i< smallersize ; i++){
        matx[i][i]= a; // fill diagonal
    }
}
```

A.10 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.11 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.12 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.13 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.14 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.15 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;
}
```