Project 4
SF2565 Program construction in C++ for Scientific Computing

Caroline Eriksson

January 2020

Part 1

The goal of this project is to use the Domain class implemented in Project 3 in order to create basic operators for enabling the solution of partial differential equations. The function to be used in the computations is:

$$u(x, y) = sin((x/10)^2)cos(x/10) + y \tag{1}$$

To begin with the Domain class created in Project 3 has to be redesigned by using smart pointers instead of regular pointers. The final Domain class header file and source file are attached in Appendix 2 and 3. The Curvebase and Matrix classes used are the same as from Project 3 and are thus not included as appendices in this report.
It is however not appropriate to use our previously created Matrix class since grids do not allow any algebraic manipulation, and a new class called GFkt shall thus be constructed.
The class GFkt will be used for grid functions defined on a certain discretized domain. Addition, multiplication with a scalar, discrete differential operators ($\partial u/\partial x$ and $\partial u/\partial y$) as well as the Laplacian Δ are some of the operations to be implemented.

The principle of the finite difference method is that derivatives in the partial differential equation are approximated by linear combinations of function values at the grid points. The equations when using the central difference method for first order differentials are:

$$\frac{\partial u_{i,j}}{\partial x} = \frac{u_{i+1,j} - u_{i-1,j}}{2\Delta x} + O(\Delta x^2) \tag{2}$$

$$\frac{\partial u_{i,j}}{\partial y} = \frac{u_{i,j+1} - u_{i,j-1}}{2\Delta y} + O(\Delta y^2) \tag{3}$$

Where Δx is the stepsize in the x-direction equal to $(5 - (-10))/50 = 0.3$, when using the same boundaries and amount of gridpoints as in Project 3. Hence the stepsize in the y-direction, Δy, is $(3 - 0)/20 = 0.15$. Using the central difference method for second order differentials the equations are:

$$\frac{\partial^2 u_{i,j}}{\partial x^2} = \frac{u_{i+1,j} - 2u_i + u_{i-1,j}}{\Delta x^2} + O(\Delta x^2) \tag{4}$$

$$\frac{\partial^2 u_{i,j}}{\partial y^2} = \frac{u_{i,j+1} - 2u_i + u_{i,j-1}}{\Delta y^2} + O(\Delta y^2) \tag{5}$$

Finally the Laplacian is naturally:

$$\Delta = \frac{\partial^2 u_{i,j}}{\partial x^2} + \frac{\partial^2 u_{i,j}}{\partial y^2} \tag{6}$$

Part 2

After creating the GFkt class and all of the necessary operations a domain is created and the grid displayed in Figure 1 below is generated. Appendix 1 presents the main source code where the domain is created.

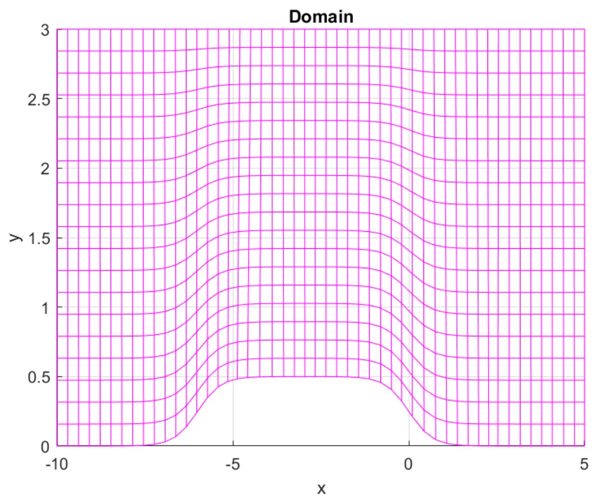

Figure 1: Chosen domain and generated grid.

The task is then to discretize the function $u(x, y)$ using the created grid and compute the differential operators (equations 2-6).
The discretization is performed by inserting the matrices for the x and y coordinates constructed in the Domain class into the function.
When trying to differentiate the grid points at the boundaries one possible source of error is the choice of method for managing the ghost points occurring. Since the function is valid for all values of x and y and there are no boundary conditions; the ghost points have to be approximated. The differential of u with regards to x is independent of y, and vice versa. Due to this and to the known stepsizes the value of u can be estimated. This estimated value of u is then inserted into the differential operators instead of the value of u which would be a ghost point.

An example of this is when being at the left boundary (x = -10). When computing $u_{i-1,j}$ the value of y is assumed to be constant (the same as for $u_{i,j}$) and the value of x is approximated as $-10 - 0.3 = -10.3$. This ghost point is then inserted into the regular equation. See Appendices 4 and 5 for the GFkt header file and source file.

Visualization of the results

The different results of the computed differential operators as well as the ones computed in Matlab are displayed in Figures 2 - 7 below. The corresponding differences between the solutions are also presented.

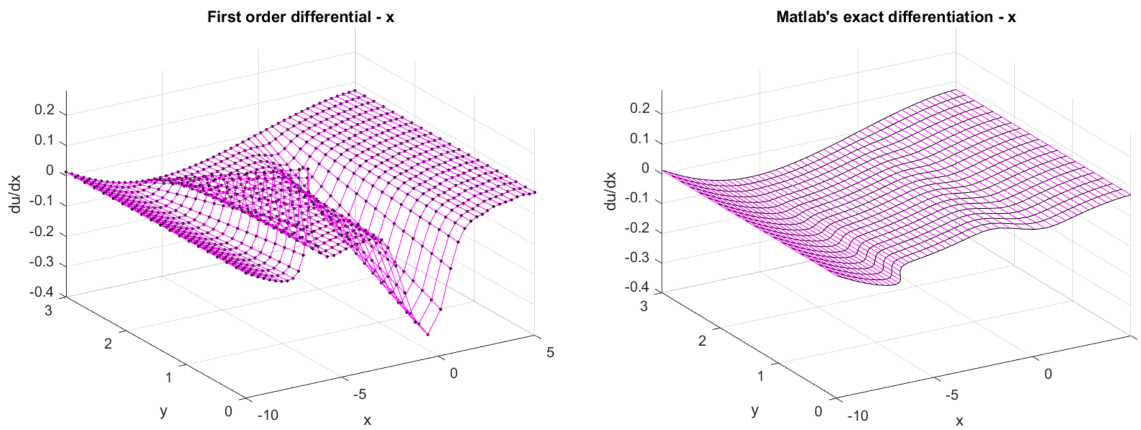

Figure 2: Computed du/dx as well as the exact one from Matlab.

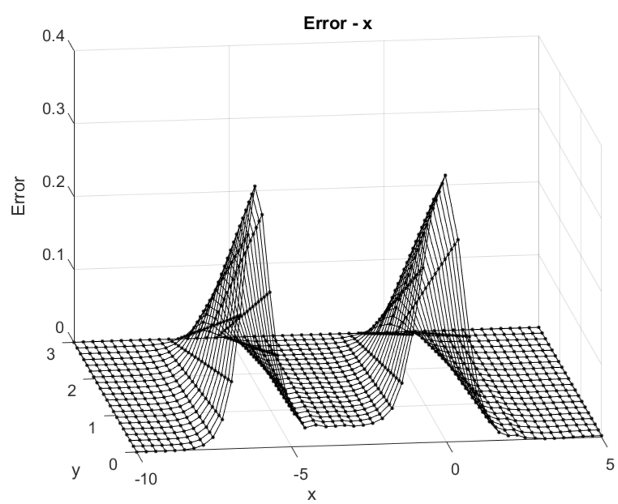

Figure 3: The difference between the solutions with regards to x.

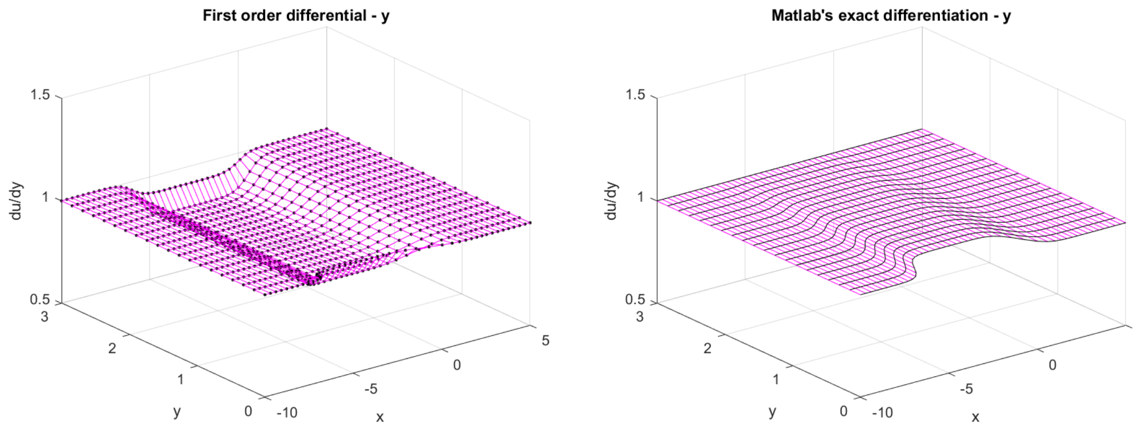

Figure 4: Computed du/dy as well as the exact one from Matlab.

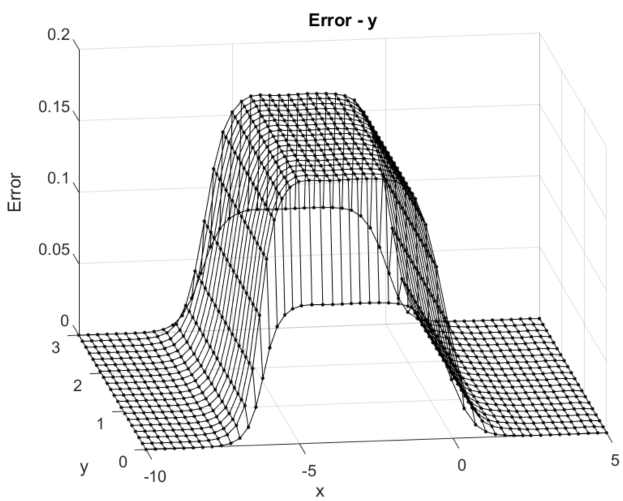

Figure 5: The difference between the solutions with regards to y.

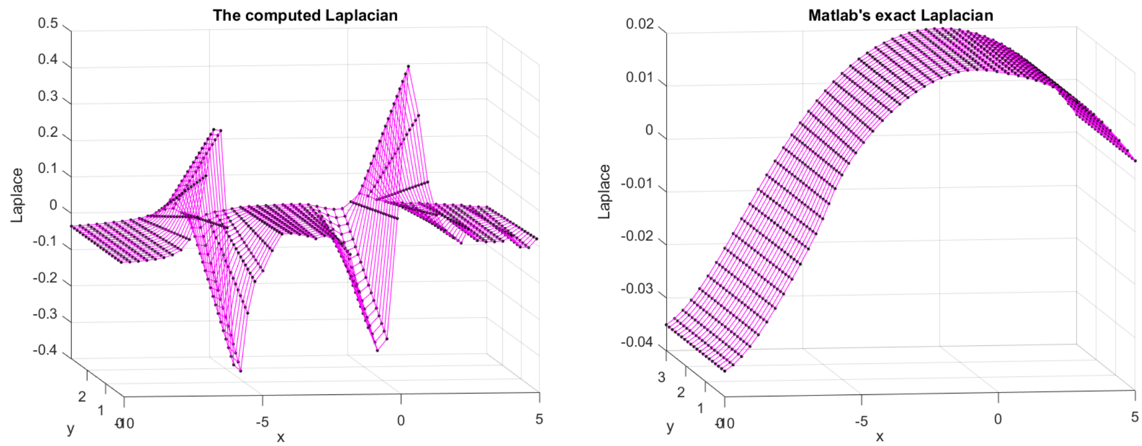

Figure 6: Computed Laplacian as well as the exact one from Matlab.

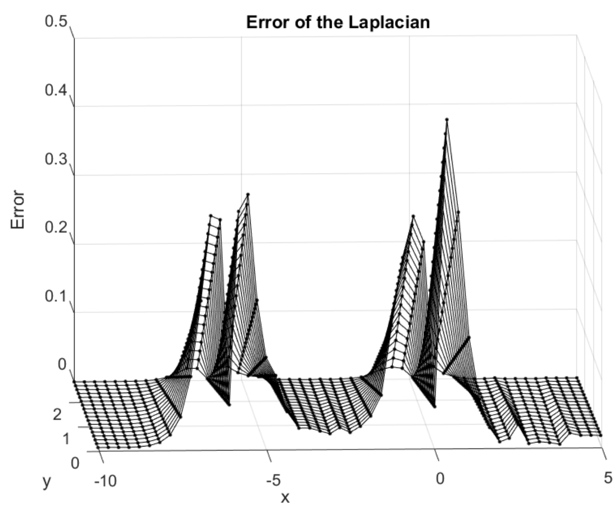

Figure 7: The difference between the solutions.

Discussion

From the images above one can see that the discrete differential operators and the Laplacian computed in this Project do not differ too much from the "exact" ones determined in Matlab. The error is largest when computing the Laplacian, which corresponds from the error for the differential with regards to x. The differential with regards to x determined with the finite difference method is however reasonable since there is a sharp gradient change at the lower boundary at approximately $x = -6$ and $x = 0$. The x-u plane of the surface is displayed in Figure 8 below.

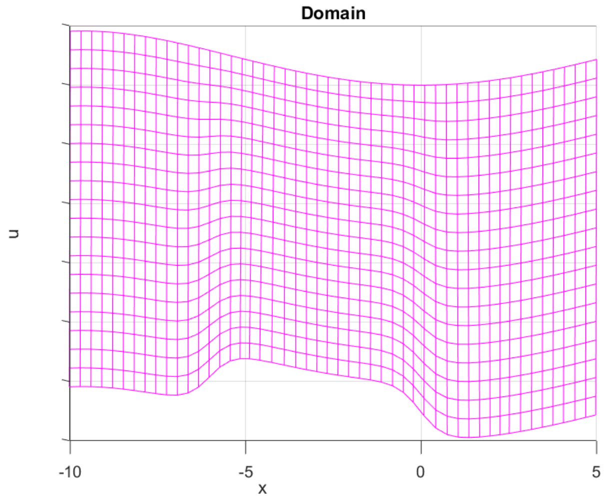

Figure 8: The surface seen from the x-u plane.

As previously mentioned one source of error might be the method used for the ghost points, since there are several ways to tackle this problem. When computing differentials in Matlab the function is first differentiated (continuously) and after that the domain is applied. By instead using the finite difference method the function values are determined to start with (by applying the domain) and then the differentials are computed.

Appendix 1: Main code

```
#include "lower_curve.h"
#include "line_v.h"
#include "line_h.h"
#include "Domain.h"
#include "Matrix.h"
#include "GFkt.h"
#include <fstream>
#include <iostream>
#include <cmath>
#include <math.h>
using namespace std;

int main()
{
        // 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;

        // 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);

        // Creation of the domain and corresponding grid
        shared_ptr<Domain> domain = shared_ptr<Domain>(new Domain(curve_0,
        line_v1, line_h2, line_v3));
        domain->grid_generation();

        // Creation of the "grid function object"
        GFkt gf = GFkt(domain);

        // Discretizing the function u
        for (int i = 0; i < domain->n + 1; i++) {
                for (int j = 0; j < domain->m + 1; j++) {
                        gf.u.Mat[i][j] = sin(pow(((domain->x[i][j]) / 10), 2))...
                        * cos((domain->x[i][j]) / 10) + domain->y[i][j];
                }
        }

        // Creation of the first order derivatives
        GFkt d_x = gf.D0x();
        GFkt d_y = gf.D0y();

        // Creation of the second order derivatives
        GFkt d2_x = gf.D2x();
        GFkt d2_y = gf.D2y();

        // Write the grid, the derivatives and the laplacian to files
        domain->writetofile();
        d_x.writetofile_dx();
        d_y.writetofile_dy();
```

```
        GFkt::writetofile_laplace(d2_x, d2_y, d2_x.grid);

        return 0;
}
```

Appendix 2: Header file for the Domain class

```
#include "line_h.h"
#include "line_v.h"
#include "lower_curve.h"
#include <memory>
using namespace std;

#ifndef DOMAIN_N
#define DOMAIN_N
class Domain
{
public:
        // Grid intervals
        const static int n = 49, m = 19;
        // Interpolated grid points
        double x[n + 1][m + 1], y[n + 1][m + 1];
        Domain(lower_curve& curve_0, line_v& line_v1, line_h& line_h2, line_v& line_v3);
        // Function generating grid
        void grid_generation();
        int xsize() { return n; }
        int ysize() { return m; }
        //Point operator()(int i, int j);
        bool grid_valid() {
                if (n != 0 && m != 0) { return true; }
        }
        // Function writing the grid to a file
        void writetofile();
private:
        // Grid points for boundary curve 0
        lower_curve curve_0;
        shared_ptr<double[]> x_0;
        shared_ptr<double[]> y_0;
        // Grid points for boundary curve 1
        line_v line_v1;
        shared_ptr<double[]> x_1;
        shared_ptr<double[]> y_1;
        // Grid points for boundary curve 2
        line_h line_h2;
        shared_ptr<double[]> x_2;
        shared_ptr<double[]> y_2;
        // Grid points for boundary curve 3
        line_v line_v3;
        shared_ptr<double[]> x_3;
        shared_ptr<double[]> y_3;
};
#endif
```

Appendix 3: Source code for the Domain class

```
#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 = shared_ptr<double[]>(new double[n + 1]);
        y_0 = shared_ptr<double[]>(new double[n + 1]);
        // Grid points for boundary curve 1
        line_v1 = line_v(line_v1_in);
        x_1 = shared_ptr<double[]>(new double[m + 1]);
        y_1 = shared_ptr<double[]>(new double[m + 1]);
        // Grid points for boundary curve 2
        line_h2 = line_h(line_h2_in);
        x_2 = shared_ptr<double[]>(new double[n + 1]);
        y_2 = shared_ptr<double[]>(new double[n + 1]);
        // Grid points for boundary curve 3
        line_v3 = line_v(line_v3_in);
        x_3 = shared_ptr<double[]>(new double[m + 1]);
        y_3 = shared_ptr<double[]>(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() {
        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 4: Header file for the GFkt class

```
#include "Domain.h"
#include "Matrix.h"
#include <stdexcept>
#include <memory>

#ifndef GFKT_H
#define GFKT_H

class GFkt {
public:
        Matrix u;
        shared_ptr<Domain> grid;
        // Standard operations
        GFkt(shared_ptr<Domain> grid_) :
                u(grid_->xsize() + 1, grid_->ysize() + 1),
                grid(grid_) {}
        GFkt(const GFkt& U) : u(U.u), grid(U.grid) {}
        GFkt& operator = (const GFkt & U);
        GFkt operator+(const GFkt& U) const;
        GFkt operator*(const GFkt& U) const;
        // First and second order differentiations
        GFkt D0x() const;
        GFkt D0y() const;
        GFkt D2x() const;
        GFkt D2y() const;
        // Writing to files
        void writetofile_dx();
        void writetofile_dy();
        static void writetofile_laplace(GFkt& d2_x, GFkt& d2_y, shared_ptr<Domain> grid);
};

#endif
```

Appendix 5: Source code for the GFkt class

```
#include "GFkt.h"
#include "Domain.h"
#include "Matrix.h"
#include <iostream>
#include <fstream>

GFkt& GFkt::operator=(const GFkt& U) {
        if (this == &U) {
                return(*this);
        }
        this->u = U.u;
        this->grid = U.grid;
        return (*this);
}

GFkt GFkt::operator+(const GFkt& U) const {
        if (grid == U.grid) { // defined on the same grid?
                GFkt tmp(grid);
                tmp.u = u + U.u; // Matrix::operator+()
                return tmp;
        }
        else throw std::runtime_error("Not defined on the same grid.");
}
GFkt GFkt::operator*(const GFkt& U) const {
        if (grid == U.grid) { // defined on the same grid?
                GFkt tmp(grid);
                for (int j = 0; j <= grid->ysize(); j++)
                        for (int i = 0; i <= grid->xsize(); i++)
                                tmp.u.Mat[i][j] = u.Mat[i][j] * U.u.Mat[i][j];

                return tmp;
        }
        else throw std::runtime_error("Not defined on the same grid.");
}

/* First order differentiation with respect to x */
GFkt GFkt::D0x() const {
        GFkt tmp(grid);
        double v1, v2;
        double n = grid->n;
        double dx = 15 / n;
        if (grid->grid_valid()) {
                //Generating derivative in tmp
                for (int j = 0; j <= grid->ysize(); j++) {
                        for (int i = 0; i <= grid->xsize(); i++) {
                                if (i == 0) {           // Ghost points occuring
                                        v1 = sin(pow(((grid->x[i][j] - dx) / 10), 2))...
                                        * cos((grid->x[i][j] - dx) / 10) + grid->y[i][j];
                                        tmp.u.Mat[i][j] = (u.Mat[i + 1][j] - v1) / (2 * dx);
                                        continue;
                                }
                                if (i == grid->xsize()) {       // Ghost points occuring
                                        v2 = sin(pow(((grid->x[i][j] + dx) / 10), 2))...
                                        * cos((grid->x[i][j] + dx) / 10) + grid->y[i][j];
```

```
                                        tmp.u.Mat[i][j] = (v2 - u.Mat[i - 1][j]) / (2 * dx);
                                        continue;
                                }
                                // Inside the boundaries where no ghost points occurs
                                tmp.u.Mat[i][j] = (u.Mat[i + 1][j] - u.Mat[i - 1][j])...
                                / (2 * dx);
                        }
                }
        }
        return tmp;
}

/* First order differentiation with respect to y */
GFkt GFkt::D0y() const {
        GFkt tmp(grid);
        double v1, v2;
        double m = grid->m;
        double dy = 3 / m;
        if (grid->grid_valid()) {
                // Generating derivative in tmp
                for (int i = 0; i <= grid->xsize(); i++) {
                        for (int j = 0; j <= grid->ysize(); j++) {
                                if (j == 0) {           // Ghost points occuring
                                        v1 = sin(pow(((grid->x[i][j]) / 10), 2))...
                                        * cos((grid->x[i][j]) / 10) + grid->y[i][j] - dy;
                                        tmp.u.Mat[i][j] = (u.Mat[i][j + 1] - v1) / (2 * dy);
                                        continue;
                                }
                                if (j == grid->ysize()) {       // Ghost points occuring
                                        v2 = sin(pow(((grid->x[i][j]) / 10), 2))...
                                        * cos((grid->x[i][j]) / 10) + grid->y[i][j] + dy;
                                        tmp.u.Mat[i][j] = (v2 - u.Mat[i][j - 1]) / (2 * dy);
                                        continue;
                                }
                                // Inside the boundaries where no ghost points occurs
                                tmp.u.Mat[i][j] = (u.Mat[i][j + 1] - u.Mat[i][j - 1])...
                                / (2 * dy);
                        }
                }
        }
        return tmp;
}

/* Second order differentiation with respect to x */
GFkt GFkt::D2x() const {
        GFkt tmp(grid);
        double v1, v2;
        double n = grid->n;
        double dx = 15 / n;
        if (grid->grid_valid()) {
                // Generating derivative in tmp
                for (int j = 0; j <= grid->ysize(); j++) {
                        for (int i = 0; i <= grid->xsize(); i++) {
                                if (i == 0) {           // Ghost points occuring
                                        v1 = sin(pow(((grid->x[i][j] - dx) / 10), 2))...
                                        * cos((grid->x[i][j] - dx) / 10) + grid->y[i][j];
                                        tmp.u.Mat[i][j] = (u.Mat[i + 1][j]...
```

```
                                - 2*u.Mat[i][j] + v1) / (pow(dx,2));
                                continue;
                        }
                        if (i == grid->xsize()) {       // Ghost points occuring
                                v2 = sin(pow(((grid->x[i][j] + dx) / 10), 2))...
                                * cos((grid->x[i][j] + dx) / 10) + grid->y[i][j];
                                tmp.u.Mat[i][j] = (v2 - 2 * u.Mat[i][j]...
                                + u.Mat[i - 1][j]) / (pow(dx, 2));
                                continue;
                        }
                        // Inside the boundaries where no ghost points occurs
                        tmp.u.Mat[i][j] = (u.Mat[i + 1][j] - 2 * u.Mat[i][j]...
                        + u.Mat[i - 1][j]) / (pow(dx, 2));
                }
            }
        }
        return tmp;
}

/* Second order differentiation with respect to y */
GFkt GFkt::D2y() const {
        GFkt tmp(grid);
        double v1, v2;
        double m = grid->m;
        double dy = 3 / m;
        if (grid->grid_valid()) {
                // Generating derivative in tmp
                for (int i = 0; i <= grid->xsize(); i++) {
                        for (int j = 0; j <= grid->ysize(); j++) {
                                if (j == 0) {           // Ghost points occuring
                                        v1 = sin(pow(((grid->x[i][j]) / 10), 2))...
                                        * cos((grid->x[i][j]) / 10) + grid->y[i][j] - dy;
                                        tmp.u.Mat[i][j] = (u.Mat[i][j + 1]...
                                        - 2 * u.Mat[i][j] + v1) / (pow(dy, 2));
                                        continue;
                                }
                                if (j == grid->ysize()) {       // Ghost points occuring
                                        v2 = sin(pow(((grid->x[i][j]) / 10), 2))...
                                        * cos((grid->x[i][j]) / 10) + grid->y[i][j] + dy;
                                        tmp.u.Mat[i][j] = (v2 - 2 * u.Mat[i][j]...
                                        + u.Mat[i][j - 1]) / (pow(dy, 2));
                                        continue;
                                }
                                // Inside the boundaries where no ghost points occurs
                                tmp.u.Mat[i][j] = (u.Mat[i][j + 1] - 2 * u.Mat[i][j]...
                                + u.Mat[i][j - 1]) / (pow(dy, 2));
                        }
                }
        }
        return tmp;
}

/*Writing dx to a file */
void GFkt::writetofile_dx() {
        ofstream differential_x;
        differential_x.open("differential_x.txt");
        for (int i = 0; i < grid->xsize() + 1; i++)
```

```
        {
                for (int j = 0; j < grid->ysize() + 1; j++)
                {
                        differential_x << u.Mat[i][j];
                        differential_x << "\t";
                }
                differential_x << "\n";
        }
        differential_x.close();
};

/*Writing dy to a file */
void GFkt::writetofile_dy() {
        ofstream differential_y;
        differential_y.open("differential_y.txt");
        for (int i = 0; i < grid->xsize() + 1; i++)
        {
                for (int j = 0; j < grid->ysize() + 1; j++)
                {
                        differential_y << u.Mat[i][j];
                        differential_y << "\t";
                }
                differential_y << "\n";
        }
        differential_y.close();
};

/*Computing and writing the laplacian to a file */
void GFkt::writetofile_laplace(GFkt& d2_x, GFkt& d2_y, shared_ptr<Domain> grid) {
        GFkt tmp(grid);
        if (grid->grid_valid()) {
                // Computation of the laplacian
                for (int i = 0; i <= grid->xsize(); i++) {
                        for (int j = 0; j <= grid->ysize(); j++) {
                                tmp.u.Mat[i][j] = d2_x.u.Mat[i][j] + d2_y.u.Mat[i][j];
                        }
                }
        }
        // Writing the laplacian to a file
        ofstream laplace;
        laplace.open("laplace.txt");
        for (int i = 0; i <= grid->xsize(); i++)
        {
                for (int j = 0; j <= grid->ysize(); j++)
                {
                        laplace << tmp.u.Mat[i][j];
                        laplace << "\t";
                }
                laplace << "\n";
        }
        laplace.close();
};
```