Project 4
Program construction in C++ for Scientific Computing HT22

Henry Jacobson

September 26, 2022

1 Task 1

I've now used smart pointers, specifically `std::shared_ptr` in the domain class for the curves.

2 Task 2

The declaration of the `GFkt` class can be found in `grid_functions.h` and the definitions in the files `grid_functions.cpp`, and `differentiation.cpp`.

To implement the laplacian we need to take the second derivative. We can either do the first derivative twice or derive the specific solution to the second derivative, I did both to compare, so here is the chain rule written out for the second derivatives.

$$\frac{\partial^2 u}{\partial \xi^2} = \frac{\partial^2 u}{\partial x^2}\left(\frac{\partial x}{\partial \xi}\right)^2 + \frac{\partial^2 u}{\partial y^2}\left(\frac{\partial y}{\partial \xi}\right)^2 + \frac{\partial^2 x}{\partial \xi^2}\frac{\partial u}{\partial x} + \frac{\partial^2 y}{\partial \xi^2}\frac{\partial u}{\partial y} + 2\frac{\partial^2 u}{\partial x \partial y}\frac{\partial x}{\partial \xi}\frac{\partial y}{\partial \xi} \tag{1}$$

$$\frac{\partial^2 u}{\partial \xi \partial \eta} = \frac{\partial^2 u}{\partial x^2}\frac{\partial x}{\partial \xi}\frac{\partial x}{\partial \eta} + \frac{\partial^2 u}{\partial y^2}\frac{\partial y}{\partial \xi}\frac{\partial y}{\partial \eta} + \frac{\partial^2 x}{\partial \xi \partial \eta}\frac{\partial u}{\partial x} + \frac{\partial^2 y}{\partial \xi \partial \eta}\frac{\partial u}{\partial y} + \frac{\partial^2 u}{\partial x \partial y}\left(\frac{\partial x}{\partial \xi}\frac{\partial y}{\partial \eta} + \frac{\partial x}{\partial \eta}\frac{\partial y}{\partial \xi}\right) \tag{2}$$

$$\frac{\partial^2 u}{\partial \eta^2} = \frac{\partial^2 u}{\partial x^2}\left(\frac{\partial x}{\partial \eta}\right)^2 + \frac{\partial^2 u}{\partial y^2}\left(\frac{\partial y}{\partial \eta}\right)^2 + \frac{\partial^2 x}{\partial \eta^2}\frac{\partial u}{\partial x} + \frac{\partial^2 y}{\partial \eta^2}\frac{\partial u}{\partial y} + 2\frac{\partial^2 u}{\partial x \partial y}\frac{\partial x}{\partial \eta}\frac{\partial y}{\partial \eta} \tag{3}$$

Since we have the solution of the first derivatives from earlier we can write this as a linear system too, using a more short-hand notation for the derivatives,

$$\begin{bmatrix} x_\xi^2 & 2x_\xi y_\xi & y_\xi^2 \\ x_\xi x_\eta & x_\xi y_\eta + x_\eta y_\xi & y_\xi y_\eta \\ x_\eta^2 & 2x_\eta y_\eta & y_\eta^2 \end{bmatrix} \begin{bmatrix} u_{xx} \\ u_{xy} \\ u_{yy} \end{bmatrix} = \begin{bmatrix} u_{\xi\xi} - u_x x_{\xi\xi} - u_y y_{\xi\xi} \\ u_{\xi\eta} - u_x x_{\xi\eta} - u_y y_{\xi\eta} \\ u_{\eta\eta} - u_x x_{\eta\eta} - u_y y_{\eta\eta} \end{bmatrix} \tag{4}$$

Where of course u_{xx}, u_{yy} and u_{xy} are the unknown.

When calculating the cross derivative we need to take care of the corners and edges specially. For the edges in y-direction we have

$$u_{xy} = \pm\frac{3u_y[i,j] - 4u_y[i \pm 1, j] + u_y[i \pm 2, j]}{2h_\xi} \tag{5}$$

where then we can use central difference for u_y, except for the corners where we need to use one sided difference there as well. For the edges in x-direction we then have

$$u_{xy} = u_{yx} = \pm\frac{3u_x[i,j] - 4u_x[i, j \pm 1] + u_x[i, j \pm 2]}{2h_\eta} \tag{6}$$

where we again can use central difference on the edges and one sided difference needs to be used in the corners.

It turned out when calculating the second derivatives in one direction that the central difference at the second index coincided with the one sided difference (or order 2) at the first index, so I decided to use a higher order scheme:

$$u_{xx} = \frac{u[i,j] - 5u[i \pm 1, j] + 4u[i \pm 2, j] - u[i \pm 3, j]}{h_\xi^2} \tag{7}$$

3 Task 3

The partial derivatives of the specified function are, where $t = x/10$,

$$\frac{\partial f}{\partial x} = \cos\left(t^2\right)\frac{t}{5}\cos\left(t\right) - \frac{1}{10}\sin\left(t^2\right)\sin\left(t\right) \tag{8}$$

$$\frac{\partial f}{\partial y} = 1 \tag{9}$$

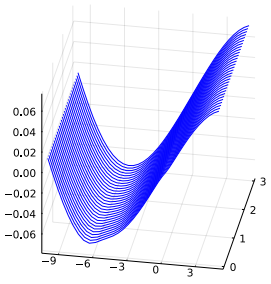
$$\nabla^2 f = \frac{-100t\sin\left(t\right)\cos\left(t^2\right) - \left(\cos\left(t\right)\left(\left(100t^2 + 25\right)\sin\left(t^2\right) - 50\cos\left(t^2\right)\right)\right)}{2500} \tag{10}$$

For an initial error estimate I used the norm functions defined in the matrix class for the total error of the derivatives. I have also plotted the solutions using the two grids defined in project 3, using 35 gridpoints in x-direction and 30 in y-driection, see figures below. I used Julia for the plotting.

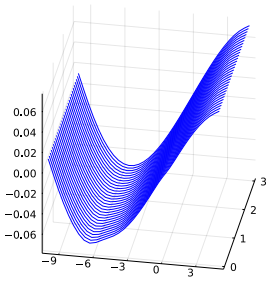
(a) Uniform grid

(b) Stretched grid

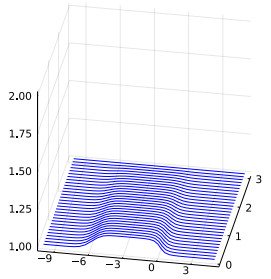
Figure 1: The function defined on the domain



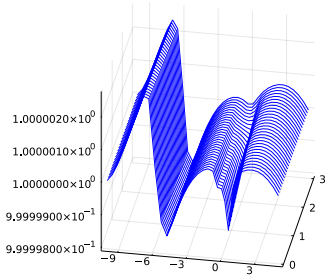
(a) Exact u_x



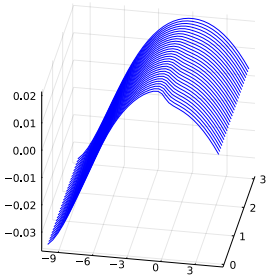
(b) Approximated u_x



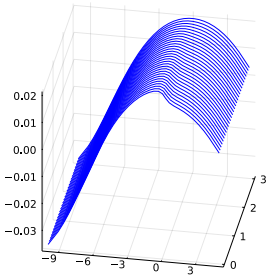
(c) Exact u_y



(d) Approximated u_y

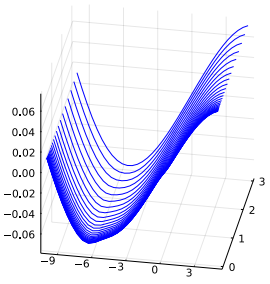


(e) Exact $\nabla^2 u$

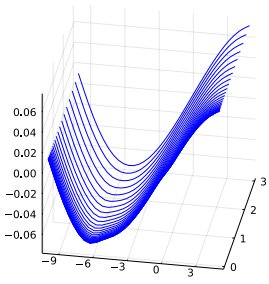


(f) Approximated $\nabla^2 u$

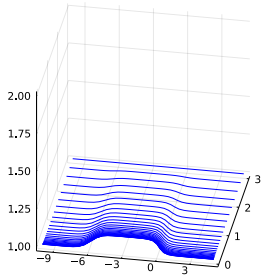
Figure 2: Analytic vs. Approximate solutions



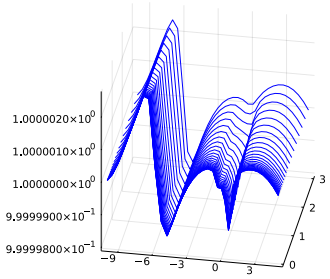
(a) Exact u_x



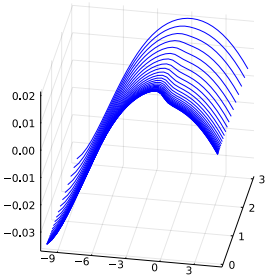
(b) Approximated u_x



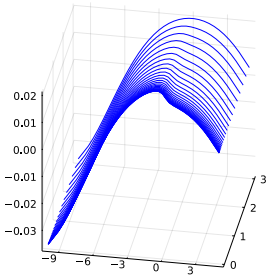
(c) Exact u_y



(d) Approximated u_y



(e) Exact $\nabla^2 u$



(f) Approximated $\nabla^2 u$

Figure 3: Analytic vs. Approximate solutions on the stretched grid

4 Appendix

4.1 curvebase.h

```
#ifndef CURVEBASE
#define CURVEBASE

#include "point.h"

#include <functional>

class CurveBase {
    protected :
        double pmin;
        double pmax;
        double length;
        bool rev;

        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 integrand(double q);
        double integrate(double p);
        double find_p(double s);
    public :
        CurveBase(double p_min=0, double p_max=1, bool rev=false);
        virtual ~CurveBase() {};
        Point xy(double s); // To get x/y do .first/.second
        Point get_corner(bool start);
        bool is_reversed();
        void print_corners();
};

typedef std::function<double(double)> Fnc1D;
double newton(Fnc1D f, Fnc1D df, double x0=0, double tol=1e-12, double maxit=1000);
#endif
```

4.2 curvebase.cpp

```
#include "curvebase.h"
#include "../../project1/src/adaptive_integration.h"

#include <cmath>
#include <iostream>
#include <functional>

CurveBase::CurveBase(double p_min, double p_max, bool _rev) {
    pmin = p_min; pmax = p_max; rev = _rev;
}

// Integrand function for use in Newtons algorithm as
// we need the derivative of the integral.
double CurveBase::integrand(double q) {
    double dxp_q = dxp(q); double dyp_q = dyp(q);
    return std::sqrt(dxp_q*dxp_q + dyp_q*dyp_q);
}
```

```
double CurveBase::integrate(double p) {
    // Same as the member functions as I couldnt pass the
    // member functions to the ASI function
    auto integrand = [&](double q) -> double {
        double dxp_q = dxp(q); double dyp_q = dyp(q);
        return std::sqrt(dxp_q*dxp_q + dyp_q*dyp_q);
    };
    return ASI(integrand,pmin,p);
}

double CurveBase::find_p(double s) {
    std::function<double(double)> f = [&](double p) -> double {
        return integrate(p) - s*integrate(pmax);
    };
    std::function<double(double)> df = [&](double p) -> double {
        return integrand(p);
    };
    return newton(f, df);
}

Point CurveBase::xy(double s) {
    double p = find_p(s);
    return Point(xp(p),yp(p));
}

// Returns the value of the specific corner, start of the curve if
// true, otherwise the end.
Point CurveBase::get_corner(bool start) {
    if (rev != start) {return xy(1);} else {return xy(0);}
}

bool CurveBase::is_reversed() {
    return rev;
}

void CurveBase::print_corners() {
    Point xy0 = xy(0); Point xy1 = xy(1);
    xy0.print();
    std::cout << " -> ";
    xy1.print('\n');
}

typedef std::function<double(double)> Fnc1D;
double newton(Fnc1D f, Fnc1D df, double x0, double tol, double maxit) {
    int i=0;
    double err=1;
    double x1;
    while (err > tol && i < maxit) {
        x1 = x0 - f(x0)/df(x0);
        err = fabs(x1-x0);
        x0 = x1;
        i++;
    }
    if (err < tol) {return x0;}
    std::cerr << "No convergence" << std::endl;
```

```
    exit(1);
}
```

4.3 domain.h

```
#include "curvebase.h"

#include <cmath>
#include <memory>

enum StretchDir {
    STRETCH_X, STRETCH_Y
};

class Domain {
    static double phi0(double s) {return 1-s;}
    static double phi1(double s) {return s;}
    private:
        std::shared_ptr<CurveBase> borders[4];
        int n,m;
        std::unique_ptr<double[]> x,y;
        std::function<double(double)> stretch_x, stretch_y;
        bool check_consistency(); // Checks that all the curves end where the next starts.
    public:
        Domain(std::shared_ptr<CurveBase>, std::shared_ptr<CurveBase>,
                std::shared_ptr<CurveBase>, std::shared_ptr<CurveBase>);
        Domain(const Domain&);
        Domain& operator=(const Domain&);
        ~Domain() {};
        Point operator()(int, int);

        void set_stretching(std::function<double(double)>, StretchDir);
        void generate_grid(int n_, int m_);
        void print_corners();

        int xsize();
        int ysize();
        double getx(int, int);
        double gety(int, int);

        void save_boundary(const char*, int precision=50);
        void save_grid(const char*);
};
```

4.4 domain.cpp

```
#include "domain.h"

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

bool Domain::check_consistency() {
    Point first,second;
    for (int i=0; i<4; i++) {
        first = borders[i]->get_corner(true); second = borders[(i+1)%4]->get_corner(false);
        if (!(eps_equal(first, second, 1e-5))) {
```

```
            return false;
        }
    }
    return true;
}

Domain::Domain(std::shared_ptr<CurveBase> c0, std::shared_ptr<CurveBase> c1,
        std::shared_ptr<CurveBase> c2, std::shared_ptr<CurveBase> c3) {
    n = 0; m = 0; x = nullptr; y = nullptr;
    borders[0] = c0; borders[1] = c1;
    borders[2] = c2; borders[3] = c3;

    stretch_x = [](double t) -> double {return t;};
    stretch_y = [](double t) -> double {return t;};

    if (!check_consistency()) {
        std::cout << "Boundary curves does not form a closed surface." << std::endl;
        for (int i=0; i<4; i++) {
            borders[i] = nullptr;
        }
        exit(1);
    }
}

Domain::Domain(const Domain& D) : n(D.n), m(D.m), x(nullptr), y(nullptr),
        stretch_x(D.stretch_x), stretch_y(D.stretch_y) {
    for (int i=0; i<4; i++) {
        borders[i] = D.borders[i];
    }
    if (m > 0) {
        x = std::make_unique<double[]>(n*m);
        y = std::make_unique<double[]>(n*m);
        for (int i=0; i<n*m; i++) {
            x[i] = D.x[i];
            y[i] = D.y[i];
        }
    }
}

Domain& Domain::operator=(const Domain& D) {
    if (this != &D) {
        if (n*m != D.n*D.m) {
            if (x != nullptr || y != nullptr) {
                x = nullptr; y = nullptr;
            }
            if (D.x != nullptr || D.y != nullptr) {
                x = std::make_unique<double[]>(D.n*D.m);
                y = std::make_unique<double[]>(D.n*D.m);
            }
        }
        n = D.n;
        m = D.m;
        for (int i=0; i<n*m; i++) {
            x[i] = D.x[i];
            y[i] = D.y[i];
        }
```

```
    }
    return *this;
}

Point Domain::operator()(int i, int j) {
    if (i < 0 || i >= n || j < 0 || j >= m) {
        std::cerr << "Index out of range." << std::endl;
        exit(1);
    }
    return Point(x[i*m+j],y[i*m+j]);
}

void Domain::set_stretching(std::function<double(double)> s, StretchDir dir) {
    switch(dir) {
        case(STRETCH_X): {
            stretch_x = s;
            break;
        }
        case(STRETCH_Y): {
            stretch_y = s;
            break;
        }
    }
}

void Domain::generate_grid(int n_, int m_) {
    if ( n_ < 0 || m_ < 0 ) {
        std::cerr << "Grid generation with negative values not allowed." << std::endl;
        exit(1);
    }

    n = n_; m = m_;
    x.reset(new double[n*m]);
    y.reset(new double[n*m]);

    Point edges;
    Point corners;
    Point xy;

    Point *borders0 = new Point[n];
    Point *borders1 = new Point[m];
    Point *borders2 = new Point[n];
    Point *borders3 = new Point[m];

    double *xis = new double[n];
    double *etas = new double[m];

    // Calculate the xy(eta) and xy(xi) to reduce time spent calculating them
    // in the nested for loop
    for (int i=0; i<n; i++) {
        xis[i] = stretch_x((double)i/(n-1)); // -1 to get to 1
        borders0[i] = (*borders[0]).xy(xis[i]);
        borders2[i] = (*borders[2]).xy(xis[i]);
    }
    for (int j=0; j<m; j++) {
        etas[j] = stretch_y((double)j/(m-1));
```

```
        /* if (stretching) { etas[j] = stretch(etas[j]); } */
        borders1[j] = (*borders[1]).xy(etas[j]);
        borders3[j] = (*borders[3]).xy(etas[j]);
    }

    // Use algebraic grid generation to generate grid using interpolation
    for (int i=0; i<n; i++) {
        for (int j=0; j<m; j++) {
            edges = phi0(xis[i])*borders3[j]
                + phi1(xis[i])*borders1[j]
                + phi0(etas[j])*borders0[i]
                + phi1(etas[j])*borders2[i];
            corners = -phi0(xis[i])*phi0(etas[j])*borders0[0]
                - phi0(xis[i])*phi1(etas[j])*borders2[0]
                - phi0(etas[j])*phi1(xis[i])*borders0[n-1]
                - phi1(etas[j])*phi1(xis[i])*borders2[n-1];
            xy = edges + corners;
            x[i*m+j] = xy.x();
            y[i*m+j] = xy.y();
        }
    }

    delete[] borders0; delete[] borders1;
    delete[] borders2; delete[] borders3;
    delete[] xis; delete[] etas;
}

void Domain::print_corners() {
    for (int i=0; i<4; i++) {
        borders[i]->print_corners();
    }
}

int Domain::xsize() {
    return n;
}

int Domain::ysize() {
    return m;
}

double Domain::getx(int i, int j) {
    return x[i*m+j];
}

double Domain::gety(int i, int j) {
    return y[i*m+j];
}

void Domain::save_boundary(const char* file, int precision) {
    FILE* pfile = fopen(file, "wb");
    // Write the precision so julia can determine it explicitly
    fwrite(&precision, sizeof(int), 1, pfile);

    Point xy;
    double x,y;
```

```
    double s;
    for (int i=0; i<4; i++) {
        for (double j=0; j<=precision; j++) {
            if (borders[i]->is_reversed()) {s = 1 - j/precision;}
            else {s = j/precision;}
            xy = borders[i]->xy(s);
            x = xy.x(); y = xy.y();
            fwrite(&x, sizeof(double), 1, pfile);
            fwrite(&y, sizeof(double), 1, pfile);
        }
    }
    fclose(pfile);
}

void Domain::save_grid(const char* file) {
    FILE* pfile = fopen(file, "wb");
    // Write the shape so julia can determine it explicitly
    fwrite(&n, sizeof(int), 1, pfile);
    fwrite(&m, sizeof(int), 1, pfile);

    for (int i=0; i<n*m; i++) {
        fwrite(&x[i],sizeof(double),1,pfile);
        fwrite(&y[i],sizeof(double),1,pfile);
    }
    fclose(pfile);
}
```

4.5 grid_functions.h

```
#include "domain.h"
#include "../../project2/src/matrix.h"

#include <memory>

typedef std::function<double(Point)> Fnc2D;

enum DiffCase {
    BOTTOM_LEFT, BOTTOM_RIGHT, TOP_LEFT, TOP_RIGHT,
    LEFT, RIGHT, BOTTOM, TOP,
    INSIDE
};

enum DiffDirection1D {
    XI, ETA
};

enum DiffDirection2D {
    XI_XI, ETA_ETA, XI_ETA
};

class GFkt : std::enable_shared_from_this<GFkt> {
    public:
        GFkt() : u(), grid(nullptr), h_eta(0), h_xi(0) {}
        GFkt(std::shared_ptr<Domain>);
        GFkt(std::shared_ptr<Domain>, Matrix);
        GFkt(const GFkt& U) : u(U.u), grid(U.grid), h_eta(U.h_eta), h_xi(U.h_xi) {}
        GFkt(GFkt&&);
```

```
        GFkt& operator=(const GFkt&);
        GFkt operator+(const GFkt&) const;
        GFkt operator-(const GFkt&) const;
        GFkt operator*(const GFkt&) const;
        GFkt operator*(double) const;
        GFkt operator/(double) const;

        void fill_matrix(Fnc2D);
        void printMatrix() {u.printMatrix();}
        void save(const char*, const char* gridfile=nullptr, const char* boundaryfile=nullptr);

        std::array<std::shared_ptr<GFkt>,2> pd();
        GFkt pdx();
        GFkt pdx2();
        GFkt pdy();
        GFkt pdy2();
        GFkt laplace();
        GFkt laplace2();

        double norm_1() {return u.norm_1();}
        double norm_inf() {return u.norm_inf();}
    private:
        Matrix u;
        std::shared_ptr<Domain> grid;
        double h_eta, h_xi;

        template <class intFnc>
        double pderiv(intFnc, DiffDirection1D, int, int);
        std::shared_ptr<Matrix> pd(int, int);

        template <class intFnc>
        double pderiv2(intFnc, DiffDirection2D, int, int);
        std::shared_ptr<Matrix> pd2(int, int);

        DiffCase getDiffCase(int, int);
};
```

4.6 grid_functions.cpp

```
#include "grid_functions.h"

#include <memory>
#include <iostream>
#include <cstdio>

GFkt::GFkt(std::shared_ptr<Domain> grid_) : u(grid_->xsize(),grid_->ysize()), grid(grid_) {
    if (grid->xsize() == 0 || grid->ysize() == 0) {
        std::cerr << "Grid must be instantiated before construction of GFkt object." << std::endl;
        exit(1);
    }
    h_xi = (double)1/(grid->xsize()-1);
    h_eta = (double)1/(grid->ysize()-1);
}

GFkt::GFkt(GFkt&& U) : u(U.u), grid(U.grid), h_eta(U.h_eta), h_xi(U.h_xi) {
    U.u = Matrix();
    U.grid = nullptr;
```

```
}

GFkt& GFkt::operator=(const GFkt& U) {
    if (this != &U) {
        u = U.u;
        grid = U.grid; // Share grid
    }
    return *this;
}

GFkt GFkt::operator+(const GFkt& U) const {
    if (grid == U.grid) {
        GFkt tmp(grid);
        tmp.u = u+U.u;
        return tmp;
    } else {
        std::cerr << "Grid functions defined on different grids." << std::endl;
        exit(1);
    }
}

GFkt GFkt::operator-(const GFkt& U) const {
    if (grid == U.grid) {
        GFkt tmp(grid);
        tmp.u = u-U.u;
        return tmp;
    } else {
        std::cerr << "Grid functions defined on different grids." << std::endl;
        exit(1);
    }
}

GFkt GFkt::operator*(const GFkt& U) const {
    if (grid == U.grid) {
        GFkt tmp(grid);
        for (int i=0; i<grid->xsize(); i++) {
            for (int j=0; j<grid->ysize(); j++) {
                tmp.u(i,j) = u(i,j) * U.u(i,j);
            }
        }
        return tmp;
    } else {
        std::cerr << "Grid functions defined on different grids." << std::endl;
        exit(1);
    }
}

GFkt GFkt::operator*(double a) const {
    GFkt tmp(grid);
    tmp.u = u*a;
    return tmp;
}

GFkt GFkt::operator/(double a) const {
    GFkt tmp(grid);
    tmp.u = u/a;
```

```
    return tmp;
}

void GFkt::fill_matrix(Fnc2D f) {
    for (int i=0; i<grid->xsize(); i++) {
        for (int j=0; j<grid->ysize(); j++) {
            u(i,j) = f(grid->operator()(i,j));
        }
    }
}

void GFkt::save(const char* valfile, const char* gridfile, const char* boundaryfile) {
    if (gridfile) {
        grid->save_grid(gridfile);
    }
    if (boundaryfile) {
        grid->save_boundary(boundaryfile);
    }
    FILE* pfile = fopen(valfile, "wb");
    // Write the shape so Julia can determine it explicitly
    int n = grid->xsize(), m = grid->ysize();
    fwrite(&n, sizeof(int), 1, pfile);
    fwrite(&m, sizeof(int), 1, pfile);

    double uij;
    for (int i=0; i<n; i++) {
        for (int j=0; j<m; j++) {
            uij = u(i,j);
            fwrite(&uij,sizeof(double),1,pfile);
        }
    }
    fclose(pfile);
}
```

4.7 differentiation.cpp

```
#include "grid_functions.h"

#include <memory>
#include <iostream>

std::array<std::shared_ptr<GFkt>,2> GFkt::pd() {
    std::array<std::shared_ptr<GFkt>,2> dxy;
    dxy[0] = std::make_shared<GFkt>(grid);
    dxy[1] = std::make_shared<GFkt>(grid);
    std::shared_ptr<Matrix> pdij;
    for (int i=0; i<grid->xsize(); i++) {
        for (int j=0; j<grid->ysize(); j++) {
            pdij = pd(i,j);
            dxy[0]->u(i,j) = (*pdij)(0,0);
            dxy[1]->u(i,j) = (*pdij)(1,0);
        }
    }
    return dxy;
}

GFkt GFkt::pdx() {
```

```
    GFkt dx(grid);
    for (int i=0; i<grid->xsize(); i++) {
        for (int j=0; j<grid->ysize(); j++) {
            dx.u(i,j) = (*pd(i,j))(0,0);
        }
    }
    return dx;
}

GFkt GFkt::pdy() {
    GFkt dy(grid);
    for (int i=0; i<grid->xsize(); i++) {
        for (int j=0; j<grid->ysize(); j++) {
            dy.u(i,j) = (*pd(i,j))(1,0);
        }
    }
    return dy;
}

// Two ways to calculate the laplacian, gives same result
GFkt GFkt::laplace() {
    GFkt laplace(grid);
    std::shared_ptr<Matrix> pdd;
    for (int i=0; i<grid->xsize(); i++) {
        for(int j=0; j<grid->ysize(); j++) {
            pdd = pd2(i,j);
            laplace.u(i,j) = (*pdd)(0,0) + (*pdd)(2,0);
        }
    }
    return laplace;
}

GFkt GFkt::laplace2() {
    GFkt x(grid);
    GFkt y(grid);
    x = pdx();
    y = pdy();
    x = x.pdx();
    y = y.pdy();
    return x + y;
}

DiffCase GFkt::getDiffCase(int i, int j) {
    if (i == 0 && j == 0) return BOTTOM_LEFT;
    if (i == 0 && j == grid->ysize()-1) return TOP_LEFT;
    if (i == grid->xsize()-1 && j == 0 ) return BOTTOM_RIGHT;
    if (i == grid->xsize()-1 && j == grid->ysize()-1) return TOP_RIGHT;
    if (i == 0) return LEFT;
    if (i == grid->xsize()-1) return RIGHT;
    if (j == 0) return BOTTOM;
    if (j == grid->ysize()-1) return TOP;
    return INSIDE;
}

void set_indices(bool change, int start_index, int inds[], int length) {
    if (change) {
```

```
        for (int i=0; i<length; i++) {
            // multiply by -1 for odd indices
            inds[i] = start_index + (2*((i+1)%2)-1)*(i/2+1);
        }
    } else {
        for (int i=0; i<length; i++) {
            inds[i] = start_index;
        }
    }
}

double one_sided_diff(int factor, double s, double s1, double s2, double h) {
    // factor should be 1 for left sided difference, -1 for right sided
    return factor*(3*s - 4*s1 + s2) / (2*h);
}

double central_diff(double sp1, double sm1, double h) {
    return (sp1 - sm1) / (2*h);
}

template <class intFnc>
double one_sided_diff_2d(intFnc f, int fc1, int fc2, int is[3], int js[3], double hs[2]) {
    double s[3];
    for (int i=0; i<3; i++) {
        s[i] = one_sided_diff(fc1, f(is[i],js[0]), f(is[i],js[1]), f(is[i],js[2]), hs[1]);
    }
    return one_sided_diff(fc2, s[0], s[1], s[2], hs[0]);
}

template <class intFnc>
double one_sided_central_diff(intFnc f, DiffDirection1D dir,  int fc, int is[3], int js[3], double hs[2])
    double s[3];
    double h;
    switch(dir) {
        case(XI): {
            h = hs[1];
            for (int i=0; i<3; i++)
                s[i] = central_diff(f(is[i],js[1]), f(is[i],js[2]), hs[0]);
            break;
        } case(ETA): {
            h = hs[0];
            for (int j=0; j<3; j++)
                s[j] = central_diff(f(is[1],js[j]), f(is[2],js[j]), hs[1]);
            break;
        }
    }
    return one_sided_diff(fc, s[0], s[1], s[2], h);
}

// Implemented the same way as for second derivatives,
// not as efficient maybe, but generalizes the concept
std::shared_ptr<Matrix> GFkt::pd(int i, int j) {
    auto x = [&](int i, int j) -> double {return grid->getx(i,j);};
    auto y = [&](int i, int j) -> double {return grid->gety(i,j);};
    auto f = [&](int i, int j) -> double {return u.operator()(i,j);};
```

```
    // e: xi, n: eta (how the greek letters look)
    double xe = pderiv(x,XI,i,j), xn = pderiv(x,ETA,i,j);
    double ye = pderiv(y,XI,i,j), yn = pderiv(y,ETA,i,j);

    double ue = pderiv(f, XI, i, j);
    double un = pderiv(f, ETA, i, j);

    SquareMatrix A(2);
    double Avals[] = {
        xe, ye, xn, yn
    };
    A.fillMatrix(Avals);

    Matrix b(3,1);
    double bvals[] = {
        ue, un
    };
    b.fillMatrix(bvals);

    std::shared_ptr<Matrix> X = A.SolveEq(b);
    return X;
}

template <class intFnc>
double GFkt::pderiv(intFnc f, DiffDirection1D curr_dir, int i, int j) {
    // f is a function defined on the grid
    double h;
    int curr_ind, size;
    int ipm[4];
    int jpm[4];
    switch(curr_dir) {
        case(XI): {
            curr_ind = i; size = grid->xsize();
            set_indices(true,i,ipm,4);
            set_indices(false,j,jpm,4);
            h = h_xi;
            break;
        }
        case(ETA): {
            curr_ind = j; size = grid->ysize();
            set_indices(false,i,ipm,4);
            set_indices(true,j,jpm,4);
            h = h_eta;
            break;
        }
    }
    if (curr_ind == 0) {
        return one_sided_diff(-1, f(i,j), f(ipm[0],jpm[0]), f(ipm[2],jpm[2]), h);
    } else if (curr_ind == size-1) {
        return one_sided_diff(1, f(i,j), f(ipm[1],jpm[1]), f(ipm[3],jpm[3]), h);
    } else {
        return central_diff(f(ipm[0],jpm[0]), f(ipm[1],jpm[1]), h);
    }
}

std::shared_ptr<Matrix> GFkt::pd2(int i, int j) {
```

```
    auto x = [&](int i, int j) -> double {return grid->getx(i,j);};
    auto y = [&](int i, int j) -> double {return grid->gety(i,j);};
    auto f = [&](int i, int j) -> double {return u.operator()(i,j);};

    double xee = pderiv2(x,XI_XI,i,j), xnn = pderiv2(x,ETA_ETA,i,j);
    double yee = pderiv2(y,XI_XI,i,j), ynn = pderiv2(y,ETA_ETA,i,j);
    double xen = pderiv2(x,XI_ETA,i,j), yen = pderiv2(y,XI_ETA,i,j);

    double uee = pderiv2(f,XI_XI,i,j), unn = pderiv2(f,ETA_ETA,i,j);
    double uen = pderiv2(f,XI_ETA,i,j);

    double xe = pderiv(x,XI,i,j), xn = pderiv(x,ETA,i,j);
    double ye = pderiv(y,XI,i,j), yn = pderiv(y,ETA,i,j);

    std::shared_ptr<Matrix>uxy = pd(i,j);
    double ux = (*uxy)(0,0);
    double uy = (*uxy)(1,0);

    SquareMatrix A(3);
    double Avals[] = {
        xe*xe, 2*xe*ye,     ye*ye,
        xe*xn, xe*yn+xn*ye, ye*yn,
        xn*xn, 2*xn*yn,     yn*yn,
    };
    A.fillMatrix(Avals);

    Matrix b(3,1);
    double bvals[] = {
        uee - ux*xee - uy*yee,
        uen - ux*xen - uy*yen,
        unn - ux*xnn - uy*ynn,
    };
    b.fillMatrix(bvals);

    std::shared_ptr<Matrix> X = A.SolveEq(b);
    return X;
}

template <class intFnc>
double GFkt::pderiv2(intFnc f, DiffDirection2D curr_diff_dir, int i, int j) {
    DiffCase curr_diff_case = getDiffCase(i, j);
    double h;
    int curr_ind, size;
    int ipm[6];
    int jpm[6];
    switch (curr_diff_dir) {
        case(XI_XI) : {
            curr_ind = i;
            size = grid->xsize();
            set_indices(true,i,ipm,6);
            set_indices(false,j,jpm,6);
            h = h_xi;
            break;
        } case(ETA_ETA) : {
            curr_ind = j;
            size = grid->ysize();
```

```
            set_indices(false,i,ipm,6);
            set_indices(true,j,jpm,6);
            h = h_eta;
            break;
        } case(XI_ETA) : {
            double hs[2] = {h_xi, h_eta};
            set_indices(true,i,ipm,6);
            set_indices(true,j,jpm,6);
            switch(curr_diff_case) {
                case BOTTOM_LEFT : {
                    int is[3] = {i,ipm[0],ipm[2]}, js[3] = {j,jpm[0],jpm[2]};
                    return one_sided_diff_2d(f,-1,-1,is,js,hs);
                }
                case TOP_LEFT : {
                    int is[3] = {i,ipm[0],ipm[2]}, js[3] = {j,jpm[1],jpm[3]};
                    return one_sided_diff_2d(f,1,-1,is,js,hs);
                }
                case BOTTOM_RIGHT : {
                    int is[3] = {i,ipm[1],ipm[3]}, js[3] = {j,jpm[0],jpm[2]};
                    return one_sided_diff_2d(f,-1,1,is,js,hs);
                }
                case TOP_RIGHT : {
                    int is[3] = {i,ipm[1],ipm[3]}, js[3] = {j,jpm[1],jpm[3]};
                    return one_sided_diff_2d(f,-1,-1,is,js,hs);
                }
                case LEFT : {
                    int is[3] = {i,ipm[0],ipm[2]}, js[3] = {j,jpm[0],jpm[1]};
                    return one_sided_central_diff(f,XI,-1,is,js,hs);
                }
                case RIGHT : {
                    int is[3] = {i,ipm[1],ipm[3]}, js[3] = {j,jpm[0],jpm[1]};
                    return one_sided_central_diff(f,XI,1,is,js,hs);
                }
                case BOTTOM : {
                    int is[3] = {i,ipm[0],ipm[1]}, js[3] = {j,jpm[0],jpm[2]};
                    return one_sided_central_diff(f,ETA,-1,is,js,hs);
                }
                case TOP : {
                    int is[3] = {i,ipm[0],ipm[1]}, js[3] = {j,jpm[1],jpm[3]};
                    return one_sided_central_diff(f,ETA,1,is,js,hs);
                }
                default: {
                    return (f(ipm[0],jpm[0]) - f(ipm[0],jpm[1]) - f(ipm[1],jpm[0]) + f(ipm[1],jpm[1])) /
                        (4*h_xi*h_eta);
                }
            }
        }
    }
    double ret;
    if (curr_ind == 0) {
        ret = (2*f(i,j) - 5*f(ipm[0],jpm[0]) + 4*f(ipm[2],jpm[2]) - f(ipm[4],jpm[4])) / (h*h);
    } else if (curr_ind == size-1) {
        ret = (2*f(i,j) - 5*f(ipm[1],jpm[1]) + 4*f(ipm[3],jpm[3]) - f(ipm[5],jpm[5])) / (h*h);
    } else {
        ret = (f(ipm[0],jpm[0]) - 2*f(i,j) + f(ipm[1],jpm[1])) / (h*h);
    }
```

```
    return ret;
}
```

4.8 ludecomp.cpp

```
#include "../../project2/src/matrix.h"

#include <memory>
#include <array>

std::shared_ptr<Matrix> SquareMatrix::SolveEq(const Matrix& b) {
    std::array<std::shared_ptr<SquareMatrix>,2> LU = LUdecomp();
    std::shared_ptr<Matrix> y = LU[0]->forward_sub(b);
    std::shared_ptr<Matrix> x = LU[1]->backward_sub(*y);
    return x;
}

// TODO: Implement partial pivoting
std::array<std::shared_ptr<SquareMatrix>,2> SquareMatrix::LUdecomp() {
    std::array<std::shared_ptr<SquareMatrix>, 2> LU;
    for (int i=0; i<2; i++)
        LU[i] = std::make_shared<SquareMatrix>(n);

    std::shared_ptr<SquareMatrix> P = std::make_shared<SquareMatrix>(n,1);

    for (int i=0; i<n; i++) {
        for (int j=i; j<n; j++) {
            double sum(0);
            for (int k=0; k<i; k++) {
                sum += LU[0]->X[i*n+k] * LU[1]->X[k*n+j];
            }
            LU[1]->X[i*n+j] = X[i*n+j] - sum;
        }
        for (int j=i; j<n; j++) {
            if (i==j) {
                LU[0]->X[i*n+i] = 1;
            } else {
                double sum(0);
                for (int k=0; k<i; k++) {
                    sum += LU[0]->X[j*n+k] * LU[1]->X[k*n+i];
                }
                LU[0]->X[j*n+i] = (X[j*n+i] - sum) / LU[1]->X[i*n+i];
            }
        }
    }
    return LU;
}

std::shared_ptr<Matrix> SquareMatrix::forward_sub(const Matrix& b) {
    std::shared_ptr<Matrix> x = std::make_shared<Matrix>(n,1);
    for (int i=0; i<n; i++) {
        double sum(0);
        for (int j=0; j<i; j++) {
            sum += X[i*n+j] * (*x)(j,0);
        }
        (*x)(i,0) = (b(i,0) - sum) / X[i*n+i];
    }
```

```
    return x;
}

std::shared_ptr<Matrix> SquareMatrix::backward_sub(const Matrix& b) {
    std::shared_ptr<Matrix> x = std::make_shared<Matrix>(n,1);
    for (int i=n-1; i>=0; i--) {
        double sum(0);
        for (int j=i+1; j<n; j++) {
            sum += X[i*n+j] * (*x)(j,0);
        }
        (*x)(i,0) = (b(i,0) - sum) / X[i*n+i];
    }
    return x;
}
```

4.9 point.h

```
#ifndef POINT
#define POINT
#include <iostream>

#include <cmath>

class Point {
    private:
        double x_;
        double y_;
    public:
        Point() : x_(0), y_(0) {};
        Point(double x, double y) : x_(x), y_(y) {}
        friend std::ostream& operator<<(std::ostream& os, const Point& P);
        Point operator+(Point p) {return Point(x_+p.x_, y_+p.y_);}
        Point operator-(Point p) {return Point(x_-p.x_, y_-p.y_);}
        double x() const {return x_;}
        double y() const {return y_;}
        void print(char end='\0');
};

Point operator*(Point a, double b);
Point operator*(double a, Point b);
bool eps_equal(Point,Point,double eps=1e-10);
#endif
```

4.10 point.cpp

```
#include "point.h"

void Point::print(char end) {
    std::cout << "(" << x_ << "," << y_ << ")" << end;
}
 Point operator*(Point a, double b) {
    return Point(a.x()*b, a.y()*b);
}

Point operator*(double a, Point b) {
    return Point(b.x()*a,b.y()*a);
}
```

```cpp
bool eps_equal(Point a, Point b, double eps) {
    if (
        fabs(a.x() - b.x()) < eps
        &&
        fabs(a.y() - b.y()) < eps
    ) return true;
    else return false;
}

std::ostream& operator<<(std::ostream& os, const Point& P) {
    return os << "(" << P.x() << "," << P.y() << ")";
}
```

4.11 funcs.cpp

```cpp
#include "../src/special_curve.h"
#include "../src/straight_line.h"
#include "../src/grid_functions.h"

#include <iostream>
#include <array>
#include <memory>
#include <cmath>

void calc_and_save(bool w_stretching) {
    std::shared_ptr<Special> sp = std::make_shared<Special>();
    std::shared_ptr<StraightLine> s1 = std::make_shared<StraightLine>(0,1,5,0,0,3);
    std::shared_ptr<StraightLine> s2 = std::make_shared<StraightLine>(1,0,0,3,-10,5,true);
    std::shared_ptr<StraightLine> s3 = std::make_shared<StraightLine>(0,1,-10,0,0,3,true);

    std::shared_ptr<Domain> d = std::make_shared<Domain>(sp,s1,s2,s3); // Normal

    if (w_stretching) {
        double delta(3);
        std::function<double(double)> stretch = [delta](double sigma) -> double {
            return 1 + std::tanh(delta*(sigma-1)) / std::tanh(delta);
        };
        d->set_stretching(stretch, STRETCH_Y);
    }

    d->generate_grid(35,30);

    GFkt U(d);
    GFkt U_analyticpdx(d);
    GFkt U_analyticpdy(d);
    GFkt U_analyticlaplace(d);

    // Analytic functions
    auto fnc = [](Point P) -> double {
        double t = P.x()/10;
        return std::sin(t*t)*std::cos(t)+P.y();
    };
    auto fnc_pdx = [](Point P) -> double {
        double t = P.x()/10;
        return std::cos(t*t)*t*std::cos(t)/5 - std::sin(t*t)*sin(t)/10;
    };
```

```
auto fnc_pdy = [](Point P) -> double {
    return 1;
};
auto fnc_laplace = [](Point P) -> double {
    double t =  P.x()/10;
    return (-100*t*std::sin(t)*std::cos(t*t) - std::cos(t)*((100*t*t+25)*std::sin(t*t) -
         50*std::cos(t*t)) ) / 2500;
};

// Fill with analytic values
U.fill_matrix(fnc);
U_analyticpdx.fill_matrix(fnc_pdx);
U_analyticpdy.fill_matrix(fnc_pdy);
U_analyticlaplace.fill_matrix(fnc_laplace);

std::array<std::shared_ptr<GFkt>,2> Updxy = U.pd();
GFkt Updx = *Updxy[0];
GFkt Updy = *Updxy[1];
GFkt Ulaplace = U.laplace();

// Calculate error norms
double pdx_err = (Updx - U_analyticpdx).norm_1();
double pdy_err = (Updy - U_analyticpdy).norm_1();
double laplace_err = (Ulaplace - U_analyticlaplace).norm_1();

// Print errors
if (!w_stretching) {
    std::cout << "Uniform grid" << std::endl;
} else {
    std::cout << "Nonuniform grid" << std::endl;
}
std::cout << "------------------------" << std::endl;
std::cout << "pdx error:\t" << pdx_err << std::endl;
std::cout << "pdy error:\t" << pdy_err << std::endl;
std::cout << "laplace error:\t" << laplace_err << std::endl;
std::cout << "------------------------" << std::endl;

// Save binary files
std::string grid = "bin/data/grid";
std::string boundary = "bin/data/boundary";
std::string function = "bin/data/function";
std::string analyticpdx = "bin/data/analyticpdx";
std::string analyticpdy = "bin/data/analyticpdy";
std::string analyticlaplace = "bin/data/analyticlaplace";
std::string pdx = "bin/data/pdx";
std::string pdy = "bin/data/pdy";
std::string laplace = "bin/data/laplace";

/* std::array<std::string,9> strings = { */
std::string *strings[9] = {
    &grid,&boundary,&function,
    &analyticpdx,&analyticpdy,&analyticlaplace,
    &pdx,&pdy,&laplace
};

std::string add;
```

```
    if (!w_stretching) {
        add = ".bin";
    } else {
        add ="_stretch.bin";
    }
    for (int i=0; i<9; i++) {
        *strings[i] += add;
    }

    U.save(function.c_str(), grid.c_str(), boundary.c_str());

    U_analyticpdx.save(analyticpdx.c_str());
    U_analyticpdy.save(analyticpdy.c_str());
    U_analyticlaplace.save(analyticlaplace.c_str());

    Updx.save(pdx.c_str());
    Updy.save(pdy.c_str());
    Ulaplace.save(laplace.c_str());
}

int main() {
    calc_and_save(false);
    calc_and_save(true);
}
```

4.12 plot_function.jl

```
import Plots
import LinearAlgebra

function func_plot(gridfile, valfile)
    Z = 0;
    X = 0; Y = 0;
    open("bin/data/"*valfile*".bin") do io
        n = read(io, UInt8)
        seek(io, 4)
        m = read(io, UInt8)
        seek(io, 8)

        n = convert(Int64, n[1]); m = convert(Int64, m[1])
        Z = Array{Float64}(undef, m,n)
        read!(io, Z)
    end
    n = 0; m = 0;
    open("bin/data/"*gridfile*".bin") do io
        n = read(io, UInt8)
        seek(io, 4)
        m = read(io, UInt8)
        seek(io, 8)

        n = convert(Int64, n[1]); m = convert(Int64, m[1])
        XY = Array{Float64}(undef, 2, n*m)
        read!(io, XY)
        X = reshape(XY[1,:],m,n)
        Y = reshape(XY[2,:],m,n)
    end
    # 3D style
```

```
    p = Plots.plot();
    Plots.plot!(X',Y',Z',color="blue",label=false, camera=(10,30))
    # Plot in both directions for grid: uncomment next line
    # Plots.plot!(X,Y,Z,color="blue",label=false, camera=(10,30))
    p
end

function save_plots()
    bins = ["function","pdx","pdy","laplace","analyticpdx","analyticpdy","analyticlaplace"]
    extras = ["","_stretch"]
    for extra in extras
        for bin in bins
            bin *= extra
            p = func_plot("grid"*extra, bin)
            Plots.savefig(p, "img/"*bin*".svg")
        end
    end
end
```