SF2656 Assignment 3

GRUNEWALD Augustin

12/11/2024

Task 1 - Straight Lines and Domain Class

In this part we built the geometry classes and some of the lines. We write a class Point which will be useful in the whole assignment, and also overloaded some operators for it such as the addition, subtraction and scalar multiplication. All the code is available here.

We then derive the class StraightLine from Curve. To get our interpolated point we basically use the the two endpoints and a double s : $Point((1-s)*start.x+s*end.x, (1-s)*start.y+s*end.y);$.

For our Domain class, we use as attribute an array of four unique pointers to our Straightlines : $std :: array < std :: unique\_ptr < Curve >, 4 >$. We've also made some constant declarations for indexing our 4 sides. At this point, we also take care of the orientation of those boundary curves :

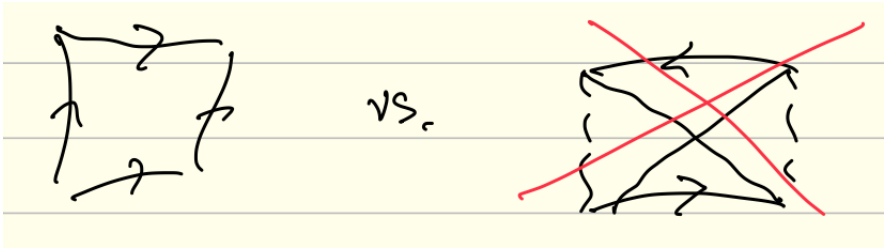

FIGURE 1 – Orientation of the boundary curves

Task 2 - Algebraic Grid Generation

We're now going to implement the algebraic grid generation using transfinite interpolation. As explained in the instruction, we have to produce as output two text files containing the coordinates of all the points of the grid. Therefore we start by implementing a twtWriter function that will write such files for each given matrices.

Secondly we implement the wanted grid generator. We start with the projectors P_ξ and P_η using linear interpolation and the tensor product and sum :

$$P_\xi[r](\xi,\eta) = (1-\xi)r(0,\eta) + \xi r(1,\eta)$$

$$P_\eta[r](\xi,\eta) = (1-\eta)r(\xi,0) + \eta r(\xi,1)$$

$$P_\xi \otimes P_\eta[r](\xi,\eta) = (1-\xi)(1-\eta)r(0,0) + (1-\xi)\eta r(0,1) + \xi(1-\eta)r(1,0) + \xi\eta r(1,1)$$

$$P_\xi \oplus P_\eta = P_\xi + P_\eta - P_\xi \otimes P_\eta$$

We use lambda functions in the code to directly build our grid. This way we get all points of the grid with

$$P_\xi \oplus P_\eta[r](ih, jh) = Point(x(ih, jh), y(ih, jh))$$

where $h = 1/n$ as said in the assignment. After filling two matrices (one for x coordinates, the other for y coordinates), we write the text files and plot the results using the supplied Python script.

We get the following grids that look correct :

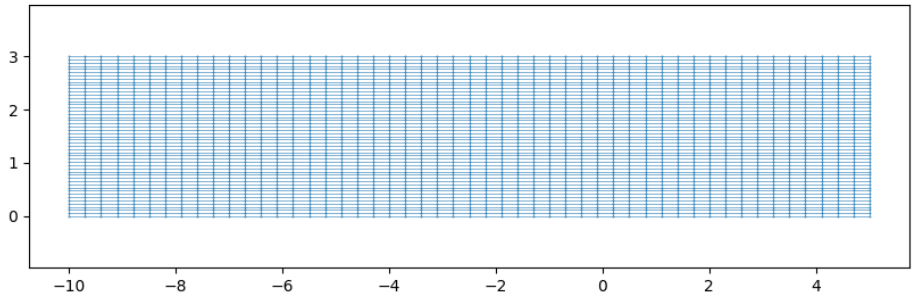

FIGURE 2 – Grid - Straight lines - n = 10

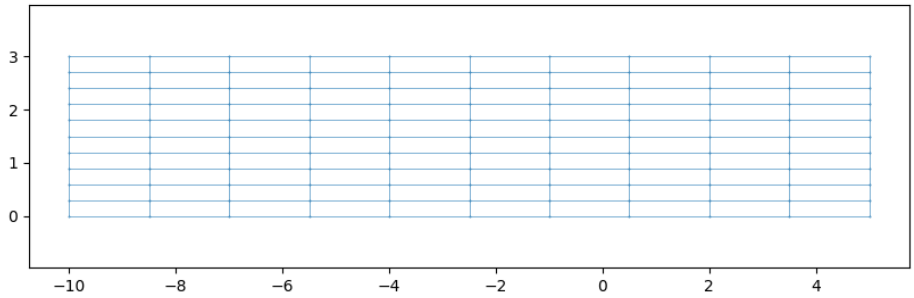

FIGURE 3 – Grid - Straight lines - n = 50

Task 3 - Bottom Curve

We will now reparameterize the bottom curve using arc length reparameterization. We start by deriving the class EquationCurve form Curve, and BottomCurve form EquationCurve. Starting with the class BottomCurve we implement the gamma and gammaPrime methods defined in EquationCurve, following the given mathematical expression.

$$f(x) = 1/(2g(x))$$

$$f'(x) = -g'(x)/(2g(x)^2)$$

$$g(x) = \begin{cases} 1 + e^{-3(x+6)}, & x \in [-10, -3], \\ 1 + e^{3x}, & x \in [-3, 5]. \end{cases}$$

$$g'(x) = \begin{cases} -3e^{-3(x+6)}, & x \in [-10, -3], \\ 3e^{3x}, & x \in [-3, 5]. \end{cases}$$

Secondly we write the EquationCurve methods. We start with a gammaPrimeNorm and an arcLength method that respectively compute the norm of the gammaPrim output Point, and the integration of the gammaPrimNorm. For the integration we use the adaptive Simpson integrator from Homework 1 which is available in *ASI.hpp*.

Finally we implement the *at* method which computes $\tilde{\gamma}(\hat{s}) = \gamma(t(\hat{s}))$. To obtain $t(\hat{s})$ we use the Newton method, which is implemented in *Newton.hpp* following this expression :

$$t_{i+1} = t_i - \frac{s(t_i) - \hat{s}s(1)}{\|\gamma'(t_i)\|}$$

we choose a starting $t_0 = \hat{s}$ which is normally always close to the wanted zero of the function, that should help for the convergence of Newton's algorithm.

Concerning the tolerances used in the ASI algorithm and in Newton's method, we choose to take both of them equal to 10^{-4}, it's a good compromise between precisions and computation times. Newton's method has also a maxIteration variable, which means that for more than 10000 iterations we get out of the algorithm loop and return the current value obtained.

Putting everything together we get the following grids. The grid generation seems to work well and the bottom curve has nicely been taken into account (to have a better look at the grid we recommend to zoom in using the matplotlib interface, these png images might be of low quality).

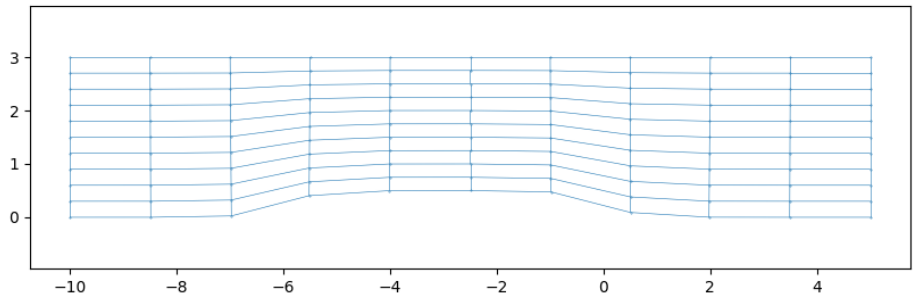

FIGURE 4 – Final Grid - n = 10

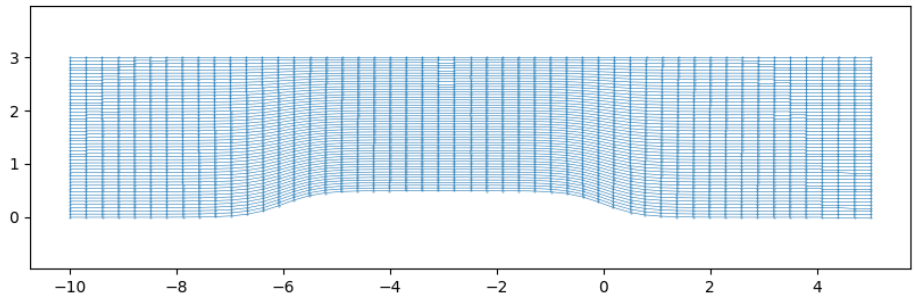

FIGURE 5 – Final Grid - n = 50

Task 4 - Point Cache

In this part we add a cache to our EquationCurve class, it enable to drastically reduce the computation time of the grid generation. We only change the *at* method that now check if the wanted value has already been computed or not.

We measure different computation times, using the *timer.hpp* from last homework, which clearly show that adding a cache is really efficient.

Number Points	**Without Cache**	**With Cache**
10	16.6514	0.669083
50	20,835.1	425.119
100	41,505.8	433.656
200	347,225	1259.51
500	5,703,750	2579.38
1000	11,407,526	6253.05

TABLE 1 – Times (ms) - Grid Generation

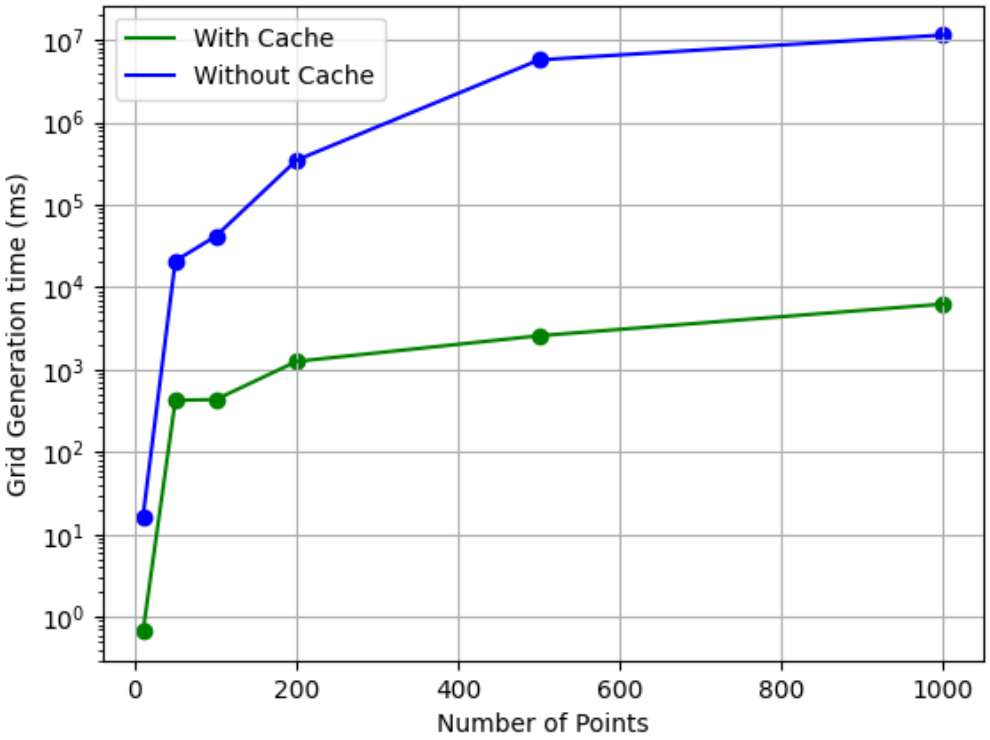

FIGURE 6 – Grid Generation times

Source code

```
#ifndef GRID_HPP
#define GRID_HPP

#include <memory>
#include <string>
#include <vector>
#include <fstream>
#include "ASI.hpp"
#include "Newton.hpp"
#include "timer.hpp"

// -- Class Point --
class Point {
public:
    double x;
    double y;

    Point(): x(0.0), y(0.0){};
    Point(double xCoord, double yCoord): x(xCoord), y(yCoord){};

    Point operator+(const Point& rhs) const {
        return Point(x + rhs.x, y + rhs.y);
    }
    Point operator-(const Point& rhs) const {
        return Point(x - rhs.x, y - rhs.y);
    }
};

inline Point operator*(double c, const Point& pt){
    return Point(c * pt.x, c * pt.y);
}
```

```
// -- Class Curve and Subclasses --
class Curve {
public:
    virtual ~Curve() = default;
    virtual Point at(double s) const = 0;
};

class StraightLine : public Curve {
public:
    StraightLine(): start(Point()), end(Point()){};
    StraightLine(Point startingPoint, Point endingPoint):
        start(startingPoint), end(endingPoint){};

    Point at(double s) const override {
        return Point((1-s) * start.x + s * end.x, (1-s) * start.y + s * end.y);
    }

private:
    Point start;
    Point end;
};

class EquationCurve : public Curve {
public:
    virtual ~EquationCurve() = default;
    Point at(double s) const override;
    double arcLength(double t) const;

private:
    virtual Point gamma(double t) const = 0;
    virtual Point gammaPrime(double t) const = 0;
    double gammaPrimeNorm(double t) const {
        return sqrt(pow(gammaPrime(t).x,2) + pow(gammaPrime(t).y,2));
    }

    mutable std::unordered_map<double, Point> cache;
};

class BottomCurve : public EquationCurve {
private:
    Point gamma(double t) const override;
    Point gammaPrime(double t) const override;
};


// -- Class Domain --
class Domain {
public:
    static constexpr int SIDE_BOTTOM = 0;
    static constexpr int SIDE_LEFT = 1;
    static constexpr int SIDE_TOP = 2;
    static constexpr int SIDE_RIGHT = 3;

    Domain(bool withBottomBoundary){
        if (withBottomBoundary) sides[SIDE_BOTTOM] =
            std::make_unique<BottomCurve>();
        else sides[SIDE_BOTTOM] = std::make_unique<StraightLine>(Point(-10,0),
            Point(5,0));

        sides[SIDE_LEFT] = std::make_unique<StraightLine>(Point(-10,0),
            Point(-10,3));
        sides[SIDE_TOP] = std::make_unique<StraightLine>(Point(-10,3),
            Point(5,3));
        sides[SIDE_RIGHT] = std::make_unique<StraightLine>(Point(5,0),
            Point(5, 3));
    }

    void algebraicGridGeneration(const std::string xPath, const std::string
        yPath, const std::size_t n) const;
```

```
    void txtWriter(const std::string path, const std::vector<double>&
        pointMatrix, const std::size_t n) const;

private:
    std::array<std::unique_ptr<Curve>, 4> sides; // bottom, left, top, right
};

#endif
```

Listing 1 – grid.hpp

```
#include "grid.hpp"

// -- Algebraic Grid Generation implementation --
void Domain::algebraicGridGeneration(const std::string xPath, const
    std::string yPath, const std::size_t n) const{
    // Setting timer
    sf::Timer timer;
    timer.start();

    if (n <= 2){
        throw std::invalid_argument("Input n should be strictly greater than
            2.");
    }

    // Initializing the matrices
    std::vector<double> xMatrix((n+1)*(n+1));
    std::vector<double> yMatrix((n+1)*(n+1));

    // Defining the projectors Pxi, PEta and the tensors product, sum
    auto Pxi = [this](double xi, double eta){
        return (1 - xi) * sides[SIDE_LEFT]->at(eta) + xi *
            sides[SIDE_RIGHT]->at(eta);
    };
    auto PEta = [this](double xi, double eta){
        return (1 - eta) * sides[SIDE_BOTTOM]->at(xi) + eta *
            sides[SIDE_TOP]->at(xi);
    };
    auto product = [this](double xi, double eta){
        return (1 - xi) * (1 - eta) * sides[SIDE_BOTTOM]->at(0) + (1 - xi) *
            eta * sides[SIDE_LEFT]->at(1) + xi * (1 - eta) *
            sides[SIDE_RIGHT]->at(0) + xi * eta * sides[SIDE_TOP]->at(1);
    };
    auto sum = [Pxi, PEta, product](double xi, double eta){
        return Pxi(xi, eta) + PEta(xi, eta) - product(xi, eta);
    };

    // Filling the matrices with the right values
    double h = 1.0 / n;
    for (std::size_t i = 0; i <= n; i++){
        for (std::size_t j = 0; j <= n; j++){
            Point gridPoint = sum(i * h, j * h);
            xMatrix[i * (n+1) + j] = gridPoint.x;
            yMatrix[i * (n+1) + j] = gridPoint.y;
        }
    }

    // Stopping timer
    timer.stop();

    // Writing the text files
    txtWriter(xPath, xMatrix, n+1);
    txtWriter(yPath, yMatrix, n+1);
}


// -- Write the matrix in a text file --
void Domain::txtWriter(const std::string path, const std::vector<double>&
    pointMatrix, const std::size_t n) const {
```

```
    std::ofstream file(path);

    for (std::size_t i = 0; i < n*n; i++){
        file << pointMatrix[i];
        if (i + 1 != n*n) ((i + 1) % n == 0)? file << "\n" : file << " ";
    }
}


// -- Equation curve implementation --
Point EquationCurve::at(double s) const {
    // Value already computed
    auto key = cache.find(s);
    if (key != cache.end()) return key->second;
    else {
        Point result = gamma(newton([this](double t){return
            gammaPrimeNorm(t);}, s, 1e-4));
        cache[s] = result;
        return result;
    }
}

// -- VERSION WITHOUT CACHE --
// Point EquationCurve::at(double s) const {
//     return gamma(newton([this](double t){return gammaPrimeNorm(t);}, s,
    1e-4));
// }

double EquationCurve::arcLength(double t) const {
    return ASI([this](double x){return gammaPrimeNorm(x);}, 0, t, 1e-4);
}

Point BottomCurve::gamma(double t) const {
    auto g = [](double x){
        return (x < -3) ? 1 + exp(-3 * (x + 6)) : 1 + exp(3 * x);
    };

    double x = (1 - t) * (-10) + 5 * t;
    double y =  1.0 / (2 * g(x));

    return Point(x,y);
}

Point BottomCurve::gammaPrime(double t) const {
    auto g = [](double x){
        return (x < -3) ? 1 + exp(-3 * (x + 6)) : 1 + exp(3 * x);
    };
    auto gPrime = [](double x){
        return (x < -3) ? -3 * exp(-3 * (x + 6)) : 3 * exp(3 * x);
    };

    double x = (1 - t) * (-10) + 5 * t;

    double xPrime = 15;
    double yPrime = - gPrime(x) * xPrime / (2 * pow(g(x),2));

    return Point(xPrime,yPrime);
}
```

Listing 2 – grid.cpp

```
#include "grid.hpp"

int main(){
    // -- DOMAIN WITH STRAIGHT LINES --
    try {
        Domain domain1(false);
        std::string xPath1 = "x1.txt";
        std::string yPath1 = "y1.txt";
```

```
        domain1.algebraicGridGeneration(xPath1, yPath1, 50);
    } catch (const std::invalid_argument& e){
        std::cerr << "Error in Domain creation : " << e.what() << std::endl;
    }

    // -- DOMAIN WITH BOTTOM CURVE --
    try {
        Domain domain2(true);
        std::string xPath2 = "x2.txt";
        std::string yPath2 = "y2.txt";
        domain2.algebraicGridGeneration(xPath2, yPath2, 50);
    } catch (const std::invalid_argument& e){
        std::cerr << "Error in Domain creation : " << e.what() << std::endl;
    }

    return 0;
}
```

Listing 3 – main.cpp

```
#ifndef ASI_HPP
#define ASI_HPP

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

inline double I(std::function<double(double)> f, double a, double b){
    return (b - a)/6 * (f(a) + 4*f((a+b)/2) + f(b));
}

inline double I2(std::function<double(double)> f, double a, double b){
    double gamma = (a + b)/2;
    return I(f, a, gamma) + I(f, gamma, b);
}

inline double ASI(std::function<double(double)> f, double a, double b, double
    tol){
    // Following the ASI algorithm
    double i_1 = I(f, a, b);
    double i_2 = I2(f, a, b);
    double errest = abs(i_1 - i_2);

    if (errest < 15*tol){
        return i_2;
    } else {
        double gamma = (a + b)/2;
        return ASI(f, a, gamma, tol/2) + ASI(f, gamma, b, tol/2);
    }
}

#endif
```

Listing 4 – ASI.hpp

```
#ifndef NEWTON_HPP
#define NEWTON_HPP

#include <functional>
#include "ASI.hpp"

inline double newton(std::function<double(double)> gammaPrimeNorm, double
    sTilde, double tol){
    double s1 = ASI(gammaPrimeNorm, 0, 1, 1e-4);
    int maxIterations = 0;

    // Initialization
    double t = sTilde;
```

```
    double tNext = t - (ASI(gammaPrimeNorm, 0, t, 1e-4) - sTilde * s1) /
        gammaPrimeNorm(t);

    while (abs(tNext - t) >= tol && maxIterations <= 1e4){
        t = tNext;
        tNext = t - (ASI(gammaPrimeNorm, 0, t, 1e-4) - sTilde * s1) /
            gammaPrimeNorm(t);
        maxIterations++;
    }
    return tNext;
}

#endif
```

Listing 5 – Newton.hpp

```
g++ -std=c++17 -Wall -o main main.cpp grid.cpp
./main
```

Listing 6 – Instruction for compiling