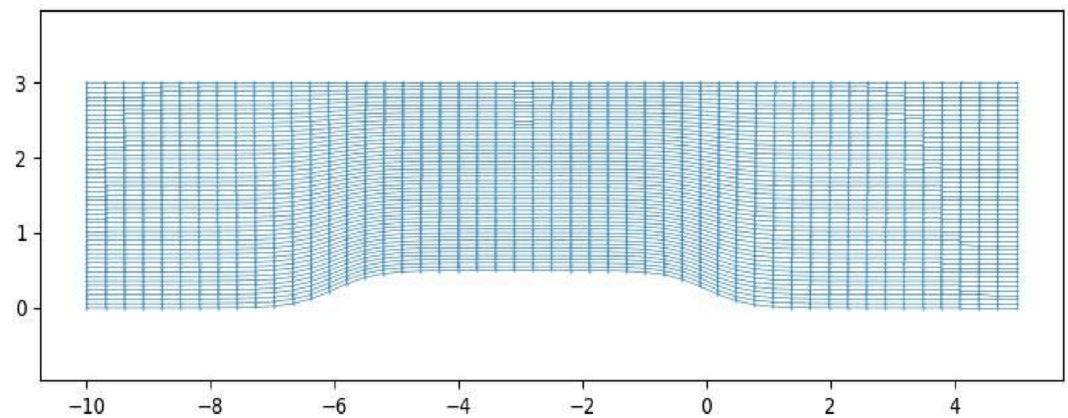
SF2565 Assignment 3 Report - Group 1

- Shawn Nagar
- Alessio Bacchiocchi

The code presented in the appendix implements a grid generator using transfinite interpolation (TFI). An example of results for N = 50 divisions along each boundary is presented below



A cache is implemented to reduce the number of point evaluations along the boundaries, and a comparison of performance is presented in the table below.
Task3 performance comparison:

Grid Divisions	Performance (msec)		Improvement (without - with) / without cache
	Without cache	**With cache**	
10	291.889	161.93	44.52%
50	7.245e3	4.276e3	40.98%
100	30.154e3	19.988e3	33.71%
200	116.420e3	71.275e3	38.78%
300	253.846	153.498e3	39.53%
500	697.024e3	440.184e3	36.85%

The result is somewhat surprising in the sense that it does not seem to scale with the number of divisions. This may be due to the way the cache is implemented and accessed for look-up.
- Perhaps data available in the cache is not reused often thus reducing the performance improvement.
- Our cache is also not aware of temporal and location based access patterns.
- Using a cache adds more overhead, especially when filling the cache at the beginning of the program.

One improvement (not implemented for reason of time) would be to store and reuse the computed value of total curve length for the bottom boundary.

Appendix - code listing

Attached below are the main code file descriptions :

1. Main code for tasks - main.cpp
2. Class definitions - class_def.hpp
3. Class implementations - class_def.cpp

main.cpp > ...

```cpp
/*
 * main.cpp
 *
 *  Created on: Nov 12, 2024
 *      Author: Shawn / Alessio
 */
#include "class_def.hpp"
#include <iostream>
#include <fstream> // For file operations
#include <chrono>
#include "timer.hpp"

using namespace std::chrono;
sf::Timer timer;

// Grid divisions: n > 2
// #define NUM_DIVISIONS 10
// #define NUM_DIVISIONS 50
// #define NUM_DIVISIONS 100
// #define NUM_DIVISIONS 150
 #define NUM_DIVISIONS 300

// (x,y)  coordinates
#define TOPLEFT -10, 3
#define TOPRIGHT 5, 3
#define BOTTOMLEFT -10, 0
#define BOTTOMRIGHT 5, 0


double bottomBoundaryFunc(double x) {

    // Weed out edge cases
    if (x < -10){
        x = -10;
    } else if (x > 5){
        x = 5;
    }

    double g = 0;
    if((x < -3) && (x ≥ -10)){
        g = 1 + exp((-3) * (x + 6));
    } else if ((x ≥ -3) && (x ≤ 5)) {
        g = 1 + exp(3*x);
    }

    return 1/(2*g);
}


void printGrid(const Grid &grid){
    // Open a file for writing
    std::ofstream filex("xdata");
    std::ofstream filey("ydata");

    // Format matrices to not have spaces for alignment of columns
    Eigen::IOFormat CommaInitFmt;
    CommaInitFmt.flags = 1;
```

main.cpp > ...

```cpp
void printGrid(const Grid &grid){

    // Print to file
    filex << grid.GetX().format(CommaInitFmt) << "\n";
    filey << grid.GetY().format(CommaInitFmt) << "\n";

    // // Print to stdout
    // std::cout << grid.GetX().format(CommaInitFmt) << "\n\n";
    // std::cout << grid.GetY().format(CommaInitFmt) << "\n";

    // Close file stream
    filex.std::ofstream::close();
    filey.std::ofstream::close();
}


// Compare performance with and without using a cache
void timeExecution(Domain& domain, bool useCache) {
    if (useCache){
        domain.enableCache(true);
        timer.start("Generating a grid with "
            + std::to_string(NUM_DIVISIONS) + " divisions, using a cache,");
    } else {
        timer.start("Generating a grid with "
            + std::to_string(NUM_DIVISIONS) + " divisions, without caching results,");
    }
    // Create Algebraic Grid
    domain.GenerateGrid();
    timer.stop();
}


int main() {
    // Create corner points
    Point topLeft(TOPLEFT);
    Point topRight(TOPRIGHT);
    Point bottomLeft(BOTTOMLEFT);
    Point bottomRight(BOTTOMRIGHT);

    // Create pointers to lines of domain
    std::unique_ptr<BottomCurve> bottom = std::make_unique<BottomCurve>(bottomBoundaryFunc);
    std::unique_ptr<StraightLine> top = std::make_unique<StraightLine>(topLeft, topRight);
    std::unique_ptr<StraightLine> left = std::make_unique<StraightLine>(bottomLeft, topLeft);
    std::unique_ptr<StraightLine> right = std::make_unique<StraightLine>(bottomRight, topRight);

    // Generate Domain
    Domain domain(std::move(bottom), std::move(top),
            std::move(left), std::move(right),
            NUM_DIVISIONS);

    // // Create Algebraic Grid
    // domain.GenerateGrid();

    // Time grid generation without and with a point cache
    timeExecution(domain, false);
    timeExecution(domain, true);
```

```

    const auto& grid = domain.GetGrid();

    // Write to file
    printGrid(grid);

    return 0;
}

```

class_def.hpp > ...

```cpp
/*
 * class_def.hpp
 *
 *  Created on: Nov 12, 2024
 *      Author: Shawn / Alessio
 */

#ifndef CLASS_DEF_HPP_
#define CLASS_DEF_HPP_

#include <cmath>
#include <vector>
#include <memory>
#include <cassert>
#include <limits>
#include <iostream>

#include "Eigen/Eigen"
#include <boost/math/quadrature/trapezoidal.hpp>
#include <boost/math/differentiation/finite_difference.hpp>


// Holds an x,y point
class Point {

public:
    // Default Constructor
    // Point();
    Point() : x(0.0), y(0.0) {}
    // Parameter Constructor
    Point(const double xCoord, const double yCoord)
    : x(xCoord), y(yCoord){}
    // Coordinate variables
    double x, y;

private:

protected:
};


// Used to represent the bottom curve
class Curve {
public:
    virtual ~Curve() = default;
    virtual Point at(double t) const = 0;

private:

protected:
};


class EquationCurve:public Curve {

```

```cpp
public:
    virtual~EquationCurve() = default;

    // Compute reparametized curve
    // Option to use point cache to compare peformance
    Point at(double t) const override;

    // Activate point cache for performance testing
    void enableCache(bool enable) {
        cacheEnabled = enable;
    }

private:
    // Compute arc-length function s(t) using boost numerical integration
    double arcLength(double t) const;

    virtual Point gamma(double t) const = 0;
    virtual Point gammaprime (double t) const = 0;

    const double TOL = 1e-12; // Tolerance for numerical calculations
    const uintmax_t MAX_ITER = 10000;    // Max iterations for newton method

protected:
    std::function<double(double)> eqFunc;
    bool cacheEnabled = false; // point cache toggle status
    mutable std::unordered_map<double, Point> cache; // Cache of already computed points
};


// Used to represent the line boundaries of the top/left/right
class StraightLine : public Curve {

public:
    StraightLine(Point a, Point b):
        pointStart(a), pointEnd(b) { }

    // t = {0,1}. t = 0 → pointStart. t = 1 → pointEnd
    Point at(double t) const override;

    Point pointStart;        // Start of line
    Point pointEnd;          // End of line

private:
    // Tolerance to check for edge cases of vertical boundaries
    static constexpr double EPSILON = std::numeric_limits<double>::epsilon();

protected:
};


class BottomCurve : public EquationCurve {

public:
    BottomCurve(std::function<double(double)> func) {
        eqFunc = func;       // Function defining curve
    }
```

class_def.hpp > EquationCurve > enableCache(bool)

```cpp

    // Domain Function of curve
    double x_of_t(double t) const;

    // Returns P(t)
    Point gamma(double t) const override;

    // Calculates dP(t)/dt using finite differences
    Point gammaprime(double t) const override;

private:
protected:
};


//Class to hold the matrices of x and y coordinates generated by
//the Domain TFI method
class Grid {

public:
    // Constructor initializes grid with given dimensions
    Grid(int rows, int cols)
    : x(Eigen::MatrixXd::Zero(rows, cols)),
      y(Eigen::MatrixXd::Zero(rows, cols))
    { }

    // Getters for the x and y matrices
    const Eigen::MatrixXd& GetX() const { return x; }
    const Eigen::MatrixXd& GetY() const { return y; }

    // Setters for the x and y matrices
    void SetX(const Eigen::MatrixXd& newX) { x = newX; }
    void SetY(const Eigen::MatrixXd& newY) { y = newY; }

    // Access element in the grid
    void SetPoint(int row, int col,
                double xValue, double yValue) {
        x(row, col) = xValue;
        y(row, col) = yValue;
    }

private:
    Eigen::MatrixXd x; // Matrix holding x-coordinates
    Eigen::MatrixXd y; // Matrix holding y-coordinates

protected:
};


class Domain {

public:
    Domain(std::unique_ptr<Curve> bottom, std::unique_ptr<Curve> top,
            std::unique_ptr<Curve> left, std::unique_ptr<Curve> right,
            int numDivisions)
    : bottom(std::move(bottom)), top(std::move(top)),
```

class_def.hpp > Domain

```cpp
    : bottom(std::move(bottom)), top(std::move(top)),
      left(std::move(left)), right(std::move(right)),
      grid(numDivisions + 1, numDivisions + 1)
    { }

    // Grid-object generator
    // Generates the grid based on transfinite interpolation (TFI)
    void GenerateGrid();
    // Generates an x, y pair based on xi and eta using TFI
    // Based on 'Basic structured grid generation (Farrashkhalvat, Miles)
    // section 4.3.2
    Point TFI(double xi, double eta);

    // Activate point cache of bottom boundary for performance testing
    void enableCache(bool enable) {
        auto* bottomBoundary = dynamic_cast<EquationCurve*>(bottom.get());
        bottomBoundary->enableCache(enable);
    }

    // Grid getter
    const Grid& GetGrid() const { return grid; }

private:
    std::unique_ptr <Curve> bottom;
    std::unique_ptr <Curve> top;
    std::unique_ptr <Curve> left;
    std::unique_ptr <Curve> right;
    Grid grid; // Grid that holds x and y coordinates

protected:
};


#endif /* CLASS_DEF_HPP_ */
```

class_def.cpp > ...

```cpp
/*
 * class_def.cpp
 *
 *  Created on: Nov 29, 2024
 *      Author: Shawn / Alessio
 */

#include "class_def.hpp"

/**EquationCurve**/
double EquationCurve::arcLength(double t) const {
    // No need to integrate if t is 0
    if (t == 0) return 0;

    // Define a lambda function
    // to generate the integrand for arc-length calculation
    auto normPdot = [this](double tau) -> double {
        Point Pdot = this->gammaprime(tau);
        return std::sqrt(Pdot.x * Pdot.x + Pdot.y * Pdot.y);
    };

    using namespace boost::math::quadrature;
    // Perform numerical integration over [0, t]
    return trapezoidal(normPdot, 0.0, t, TOL);
}

Point EquationCurve::at(double t) const {
    if (cacheEnabled) {
        // Check if P(t) has been computed previously
        auto found = cache.find(t);
        if (found != cache.end()) {
            return found->second; // Return the cached point
        }
    }

    const double hatS = t; // use reference to keep same signature as Curve class
    // while avoiding confusion in below function equations
    const double arcLengthTotal = arcLength(1.0); // full curve length
    double arcLengthTarget = hatS * arcLengthTotal; // Target arc-length value

    // Define the function f(t) = s(t) - sTarget
    auto f = [this, arcLengthTarget](double t) -> double {
        return arcLength(t) - arcLengthTarget;
    };

    // Define the function f'(t) = |dP/dt|
    auto f_prime = [this](double t) -> double {
        Point Pdot = this->gammaprime(t);
        return std::sqrt(Pdot.x * Pdot.x + Pdot.y * Pdot.y);
    };

    // Newton method to find root convergence
    double t_of_hatS = t;
    for(uintmax_t n = 0; n < MAX_ITER; n++) {
        t_of_hatS = t - f(t)/f_prime(t);

```

class_def.cpp > ...

```cpp
Point EquationCurve::at(double t) const {
    for(uintmax_t n = 0; n < MAX_ITER; n++) {
        // Upon convergence, cache and return the computed point
        if(fabs(t_of_hatS - t) < TOL) {
            Point result = gamma(t_of_hatS);
            // Store result in cache if required
            if (cacheEnabled)
                cache[t] = result;

            return result;
        }
        // update previous t value
        t = t_of_hatS;
    }
    throw std::runtime_error("Newton's method didn't converge");
}

/**StraightLine**/
Point StraightLine::at(double t) const {
    // Edge case for vertical line
    if(fabs(pointEnd.x - pointStart.x) < EPSILON) {
        Point p_toGet(pointStart.x,0);  // Init y point as 0 for now
        p_toGet.y = pointStart.y + t*(pointEnd.y - pointStart.y);
        return p_toGet;
    }// Edge case for horizontal line
    else if (fabs(pointEnd.y - pointStart.y) < EPSILON) {

        Point p_toGet(0,pointStart.y);  // Init x point as 0 for now
        p_toGet.x = pointStart.x + t*(pointEnd.x - pointStart.x);
        return p_toGet;
    } else {
        double pointOfInterest = pointStart.x + t*(pointEnd.x - pointStart.x);
        Point p_toGet(pointOfInterest,0);   // Init y point as 0 for now
        // p_toGet.y = pointStart.y + t*(pointEnd.x - pointStart.x);
        p_toGet.y = pointStart.y + t*(pointEnd.y - pointStart.y);
        return p_toGet;
    }
}

/**BottomCurve**/
double BottomCurve::x_of_t(double t) const {
    //Check for edge cases
    if (t < 0) t = 0;
    if (t > 1) t = 1;
    // Describes domain (-10,5)
    double x = (1 - t) * (-10) + 5 * t;
    return x;
}

Point BottomCurve::gamma(double t) const {
    double x = x_of_t(t);
    Point p_of_t(x, eqFunc(x));
    return p_of_t;
};

Point BottomCurve::gammaprime(double t) const {
```

class_def.cpp > ...

```cpp
Point BottomCurve::gammaprime(double t) const {
    using namespace boost::math::differentiation;

    // Calculates x-dot. Uses capture variable to access the x_of_t class method
    auto x_dot = finite_difference_derivative(
            [this](double t_val) { return x_of_t(t_val);}, t);

    // Calculates y-dot.
    auto y_dot = finite_difference_derivative(
            [this](double t_val) {
        double x = x_of_t(t_val);
        return eqFunc(x);
    }, t);

    return Point(x_dot, y_dot);
};

/**Domain**/
void Domain::GenerateGrid() {
    int divisions = grid.GetX().rows() - 1; // `numDivisions` intervals
    for (int i = 0; i <= divisions; ++i) {
        double eta = static_cast<double>(i) / divisions;
        for (int j = 0; j <= divisions; ++j) {
            double xi = static_cast<double>(j) / divisions;

            // Use TFI to compute coordinates
            Point p = TFI(xi, eta);
            grid.SetPoint(i, j, p.x, p.y);
        }
    }
}

Point Domain::TFI(double xi, double eta) {
    // Use the boundary curves to compute the x, y coordinates
    //
    auto [xBottom_atXi, yBottom_atXi] = bottom->at(xi);
    auto [xTop_atXi, yTop_atXi] = top->at(xi);
    auto [xRight_atEta, yRight_atEta] = right->at(eta);
    auto [xLeft_atEta, yLeft_atEta] = left->at(eta);
    auto [xBottom_atZero, yBottom_atZero] = bottom->at(0);
    auto [xTop_atZero, yTop_atZero] = top->at(0);
    auto [xBottom_atOne, yBottom_atOne] = bottom->at(1);
    auto [xTop_atOne, yTop_atOne] = top->at(1);

    double x = (1 - xi) * xLeft_atEta + xi * xRight_atEta
            + (1 - eta) * xBottom_atXi + eta * xTop_atXi
            - (1 - xi) * (1 - eta) * xBottom_atZero
            - (1 - xi) * eta * xTop_atZero
            - (1 - eta) * xi * xBottom_atOne
            - xi * eta * xTop_atOne;

    double y = (1 - xi) * yLeft_atEta + xi * yRight_atEta
            + (1 - eta) * yBottom_atXi + eta * yTop_atXi
            - (1 - xi) * (1 - eta) * yBottom_atZero
            - (1 - xi) * eta * yTop_atZero
            - (1 - eta) * xi * yBottom_atOne
```

```cpp
Point Domain::TFI(double xi, double eta) {
    // Use the boundary curves to compute the x, y coordinates
    //
    auto [xBottom_atXi, yBottom_atXi] = bottom->at(xi);
    auto [xTop_atXi, yTop_atXi] = top->at(xi);
    auto [xRight_atEta, yRight_atEta] = right->at(eta);
    auto [xLeft_atEta, yLeft_atEta] = left->at(eta);
    auto [xBottom_atZero, yBottom_atZero] = bottom->at(0);
    auto [xTop_atZero, yTop_atZero] = top->at(0);
    auto [xBottom_atOne, yBottom_atOne] = bottom->at(1);
    auto [xTop_atOne, yTop_atOne] = top->at(1);

    double x = (1 - xi) * xLeft_atEta + xi * xRight_atEta
            + (1 - eta) * xBottom_atXi + eta * xTop_atXi
            - (1 - xi) * (1 - eta) * xBottom_atZero
            - (1 - xi) * eta * xTop_atZero
            - (1 - eta) * xi * xBottom_atOne
            - xi * eta * xTop_atOne;

    double y = (1 - xi) * yLeft_atEta + xi * yRight_atEta
            + (1 - eta) * yBottom_atXi + eta * yTop_atXi
            - (1 - xi) * (1 - eta) * yBottom_atZero
            - (1 - xi) * eta * yTop_atZero
            - (1 - eta) * xi * yBottom_atOne
            - xi * eta * yTop_atOne;

    return Point(x, y);
}
```