PCPP18 - Project 3

Alma Andersson

November 29, 2018

1 C++ Version and Source Code access

All code provided for this project is written as to be compatible with C++11. It has been tested and compiled with the g++ GNU compiler on an Ubuntu 16.04 LTS system.

All code is made available at github where also instructions of how to best run the program is found.

2 Outline

This report is divided into three parts each describing the core elements of this project. The first part outlines how the boundary curves later used to compile a four sided domain are constructed. The second part describes how these boundary curves are assembled into a larger four sided domain. The third part finally describes how a grid is generated over the domain using *algebraic grid generation*. Only the main features of the program and classes/functions are outlined here, for more explicit information the reader is referred to the Appendix where the source code can be found.

3 Part 1 - Boundary Curve Formation

The Curvebase class

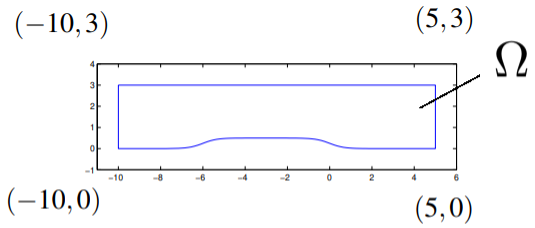
All boundary curves inherit their structure from the abstract base class *Curvebase*. This class has among it's non-virtual member the necessary functions to compute the arc-length of a curve, using adaptive simpson integration (performed using the 4 functions *integrate, function, I1 and I2*). It also contains the *solve* member function using Newton Rhapson's method to approximate the root of the expression given in equation 1,

$$s = \frac{1}{L}\int_a^{p_k} \sqrt{x'(p)^2 + y'(p)^2}dp, \qquad L = \int_a^b \sqrt{x'(p)^2 + y'(p)^2}dp \tag{1}$$

Where p is the variable used to parametrize the curve. Allowing the user to enter a "relative" position $s[0, 1]$ representing the fraction of the arc-length by which the point she wants to access is situated. As to illustrate, if $s = 0.5$ the return-value of the solve function will be the value p such that the arc-length from the lower limit to p is half of the total arc-length of the curve, note how $p \in \mathbb{R}$.

Given the parametrization $(x, y) = (x(p), y((p))$, functions to retrieve the x-respectively y-coordinate, respectively the derivatives $x'(p)$ and $y'(p)$ given p, are added as virtual functions. This, since the curve parametrization is a defining property of each individual curve, and a static parametrization would be of no use in an abstract base class.

Included in Curvebase class are also a default constructur and destructor, as well as a copy constructor, allowing for new Curvebase objects to be initialized with an old one. The Curvebase also includes a constructor, allowing for an object to be initialized with lower and upper end, points to be specified as well as curve orientation and tolerance level to be used in the numerical approximations (default is 10^{-5}). Finally the Curvebase class has a *partially virtual* function *setLength* which will compute the arc-length of a parametrized curve using Newton



The lower boundary is given by the function

$$f(x) = \begin{cases} \frac{1}{2}\frac{1}{1+exp(-3(x+6))}, & x \in [-10,-3) \\ \frac{1}{2}\frac{1}{1+exp(3x)}, & x \in [-3,5] \end{cases}$$

Figure 1: Domain Ω as defined in the assignment. (Modified picture, taken from instructions)

Rhapson's method and the equation described in eq 1; albeit a proper approach for all curves, simpler approaches for length calculations are available in certain special cases such as straight line segments allowing for unecessary computations to be avoided.

Horzline and Vertline classes

Two additional abstract base classes which inherits from the Curvebase are used representing two similar and common scenarios upon domain construction, namely where the either the y-variable is a function of the x-variable and the reverse. Allowing for parametrizations on the for (i) $(x, y) = (p, f(p))$ respectively (ii) $(x, y) = (g(p), p)$. The former scenario (i) is captured by the class *Horzline* whilst *Vertline* models the latter (ii).

Both these classes thus defines the inherited virtual functions (from Curvebase) for the x and y-coordinate parametrization and the respective derivatives. Two new virtual functions are included in each class allowing for definition of the relationship between x and y-coordinates (and the derivative) to be defined in derived classes; these classes being the *afunc(p)* and *afuncd(p)* with $a = \{x, y\}$ having the former define the coordinate mapping and the latter it's derivative.

LeftRightBorder, TopBorder and BottomBorder classes

In the main-program, three classes (LeftRightBorder, TopBorder and BottomBorder) are defined, where LeftRightBorder is a derived class of *Vertline*, handling the vertical borders that are vertical straight line segments. TopBorder as well as BottomBorder are derived classes of Horzline. TopBorder handles a scenario where we have a horizontal straight line segment, whilst BottomBorder is used for cases where the y-coordinate is a function of the x-coordinate.

The LeftRightBorder class has a member function *setXpos* that sets the static x-coordinate, whilst the member function *setYpos* in TopBorder does the equivalent but for the y-coordinate. Both the LeftRightBorder and TopBorder redefines the *getLength*-function, as to simply be the euclidian distance between upper and lower endpoint.

Boundary Curves

The following parametrizations are used for the four boundary curves in the given domain Ω (see Fig 1 for reference).

- Left Border : $(x, y) = (-10, y)$
- Bottom Border : $(x, y) = (x, f(x))$, with $f(x)$ defined in Fig 1
- Right Border : $(x, y) = (5, y)$
- Top Border : $(x, y) = (x, 3)$

4 Part 2 - Domain Formation

A class named *Domain* is used as to create the four sided domain. Given that the proper orientation of each boundary curve is set, the user is free to provide the boundary curves in any random order upon initialization of a Domain object. The function, *check_conistency* will automatically assign the boundary curves to the correct side. The directionality of a boundary curve is given by treating Ω as a positively oriented domain, and assigning those boundary curves where the flow from startpoint to endpoint is consistent with this orientation as positively oriented whilst those where the oppoiste is true are seen as negatively oriented.

Functions to access the orientation is thus included in the Curvebase base class (*ori*), also a function to return a given side of a Domain object (*getSide)* is used, as to access the boundary curves of a temporary Domain object upon sorting the domain boundaries.

5 Part 3 - Grid Generation

Once a Domain object have been initialized, the function *make_grid* allows the user to generate a structured grid with a specified number of *intervals* along the horizontal respectively vertical direction. This is done by *algebraic grid formation*, which essentially first generates a grid on the unit square $\Gamma = [0,1] \times [0,1]$ and where each point is mapped to the actual domain Ω by using linear interpolation between points known on the boundaries with either x and y-value equal to that of the point to be interpolated, as illustrated in the equation below.

$$\begin{aligned} x(s,t) = (1-s)x(0,t) + sx(1,t) + (1-t)x(s,0) + tx(s,1) \\ -s(1-t)x(1,0) - (1-s)tx(0,1) - tsx(1,1) - (1-s)(1-r)x(0,0) \end{aligned}$$

$$\begin{aligned} y(s,t) = (1-s)y(0,t) + sy(1,t) + (1-t)y(s,0) + tx(s,1) \\ -s(1-t)y(1,0) - (1-s)ty(0,1) - tsy(1,1) - (1-s)(1-r)y(0,0) \end{aligned}$$

If a grid is already present, this is erased before the rendering of a new grid. The mapping above is easily done by realizing the following

1. $x(0,t)$ and $y(0,t)$ - represents points on the left boundary curve
2. $x(1,t)$ and $y(1,t)$ - represents points on the right boundary curve
3. $x(s,0)$ and $y(s,0)$ - represents points on the bottom boundary curve
4. $x(s,1)$ and $y(s,1)$ - represents points on the top boundary curve

Given the parametrization and construction of the boundary curve classes, all these expressions can be evaluated. The negative terms representing the corners, which of course, values also are known for (given how these are the endpoints of the boundary curves. The functions to perform the mapping are named *xmap* and *ymap* respectively, and are members of the domain class.

Additional to the above mentioned member functions, the Domain has a copy constructor and assignment operator overloading, as to allow for initialization of a Domain from another Domain. This will copy all properties, including the grid (if such exists) to the new object.

As recommended in the instructions, *fwrite* in combination with *fopen* and *fclose* is used to save the generated grid as two bin-files, one for the x-coordinates and one for the y-coordinates. This is done by calling the function *saveCoordinates* a member of the Domain class.

A boolean value can be passed as a parameter to the *saveCoordinates* function, which if true will allow the user to specify the "stem" of the files to which the coordinates should be saved, to this the prefixes "x_vec_" respectively "y_vec_" will be appended. As the program is currently setup, with all source files placed in a folder "bin" the output is placed in a directory "res" positioned in the same directory as "bin". If set to false (as is default) the names *x_vec_generated_grid.bin* and *y_vec_generated_grid.bin* are used instead.

The user can also specify that the lower boundary should be higher resolved, meaning a higher density of spots as compared to the upper boundary by using the "stretching" function given in equation 2

$$T(\sigma) = 1 + \frac{\tanh(3(1-\sigma))}{\tanh(3)} \tag{2}$$

This is done by calling the function *doLowerResolve* passing "true" as an argument, if "false" is passed no adjustment of the resolution is done (this is default).

6 Visualization

In order to visualize the results, i.e. plotting the grid a simple python-script is provided, named "visualize.py", found at same level as the *res* and *bin* folders. This script takes as first argument the path to the x-coordinate file and second argument that of the y-coordinate file. This script utilize the numpy and matplotlib python libraries (which are usually included as standard packages in most python versions).

7 Results and Comments

The program *main.cpp* is constructed as a "proof of concept" , here information about all four boundary curves is printed and the functionality of the x and y member functions is demonstrated. A domain from the four boundary curves representing those in Ω is formed and two 20×50 grids are generated, one with increased lower boundary resolution and one with no adjustment to the resolution. The user is requested to enter the names of the files to which these coordinates should be saved (.bin must be the included).

Figure 2 illustrates the non-adjusted (normal) grid that was generated, visualized using the *visualize.py* script, whilst figure 3 displays the same output put for the adjusted resolution grid.

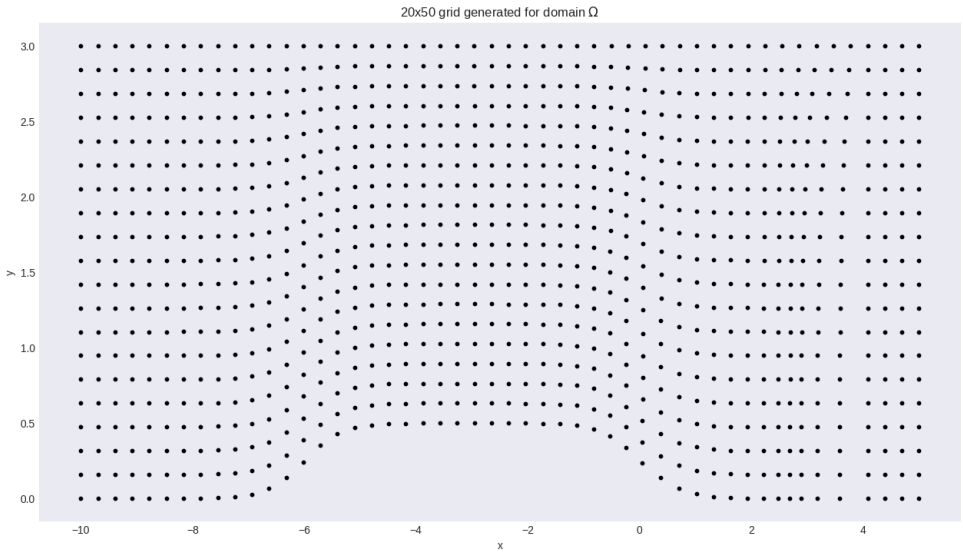


Figure 2: "Normal" grid, meaning no adjustment to the resolution is applied.

There are some artifacts to the generated grid, looking at the right side, it's possible to discern how some irregularities in the grid pattern arise. Whether this is an inherent property of the method, or a design flaw in the code was not clear. However given how the "bump" at the lower boundary slightly distorts the grid, it is of course expected

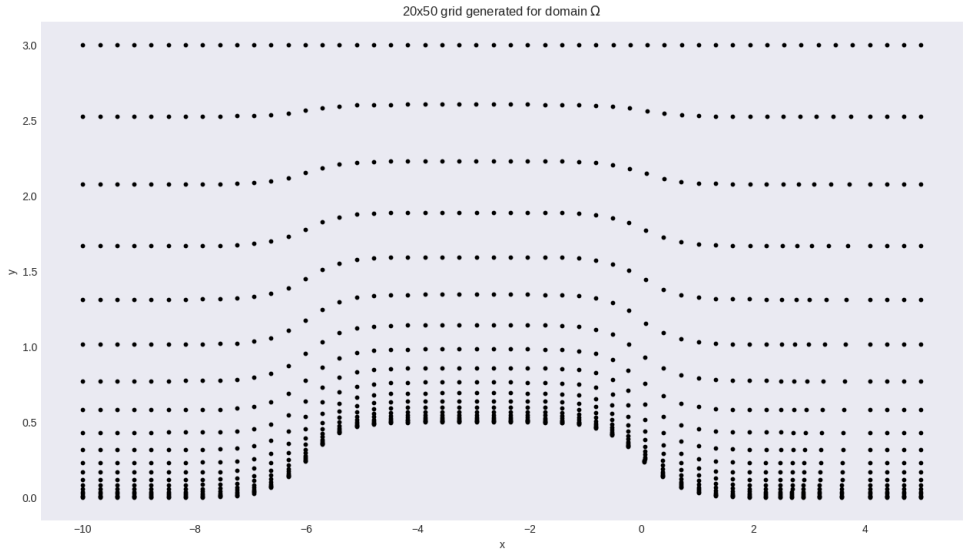


Figure 3: Adjusted resolution, with higher resolution for the lower boundary generated using by distributing the vertices in the unit-square along the vertical direction according to equation 2

that some compression and non-uniformity is seen, however the non-symmetrical effect (i.e. it is not observed on the left hand side) indicate that this might be a bug in the code.