Report
Simulation Sciences Laboratory

Minimal Surfaces

add your matriculation numbers here!
Chenfei Fan
Praveen Mishra 389850
Sankarasubramanian Ragunathan 389851
Philipp Schleich 391779

February 28, 2020

Supervisor: Prof. Dr. Uwe Naumann
Klaus Leppkes

Contents

1. Introduction

A Minimal Surface is defined as a surface that has zero mean curvature. A minimal surface is characterized as surface of minimal surface area for the given boundary conditions.For example, a plane is a trivial minimal surface.

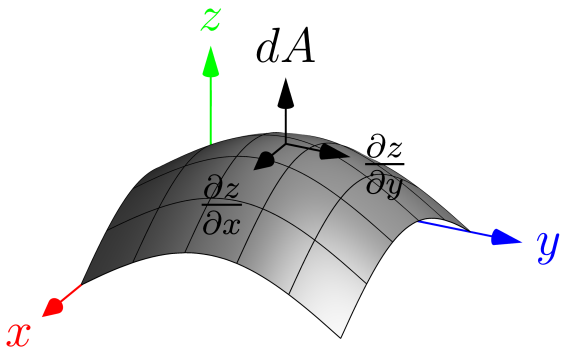

Figure 1: A *Minimal Surface* with the local area vector that is to be minimized.

The minimal surface is obtained as a solution to the Lagrange's equation (also known as the *Minimal-Surface Equation* (MSE)) given by:

$$\begin{aligned} (1+z_y^2)z_{xx} - 2z_x z_y z_{xy} + (1+z_x^2)z_{yy} = \mathcal{F}[z] = 0 \qquad & \text{in } \Omega \\ z(x,y) = g(x,y) \qquad & \text{on } \partial\Omega \end{aligned} \tag{1}$$

MSE is a Nonlinear Elliptic PDE that has a combination of first and second order differential terms in both x and y. Being a nonlinear PDE, we do not readily obtain an analytical solution necessitating the use of numerical methods, such as linearization of MSE using Newton-Raphson iterations, to solve the problem. It is also important to bear in mind that as z has second order derivative terms, we need $z \in \mathcal{C}^2$ which means that we need to provide smooth enough boundary conditions on $\partial\Omega$ in order to be able to compute the derivative terms in MSE.

MSE is used in designing structures that have great tensile strength and also to minimize the cost of construction as structural designs resulting from MSE require less material. MSE is also used to provide a relativistic description on the formation of Black Holes. Material lattice structures of biological organisms such as butterfly wings are minimal surface structures (Butterfly wings have a Gyroidal minimal surface). Hydrophobic co-polymer structures also take the shape of a minimal surface in order to reduce the energy of the structure arising due to surface tension.

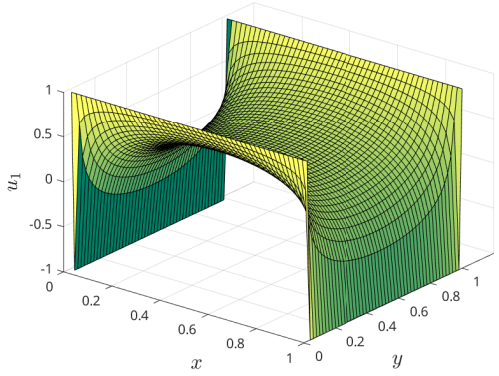

Figure 2: Scherk's first surface on $[0, 1]^2$

2. Background

2.1. General background (don't like this title)

We look at surfaces in $\mathbb{R}^3$, defined over an open set $\Omega \subset \mathbb{R}^2$. The surface of desire should contain the least possible area among all possible surfaces, that assume given values on the boundary of Ω, denoted by $\partial\Omega$. [1]

Lagrange showed in 1760, that such a surface is characterized by the graphic of a function $z(x, y)$, $z : \mathbb{R}^2 \to \mathbb{R}$, which is twice continuously differentiable on a twodimensional domain, particularly in a subset of $\mathbb{R}^2$. This function z has to fulfill the so called *Minimal-Surface Equation* (MSE) stated below.

$$\begin{aligned} (1 + z_y^2)z_{xx} - 2z_x z_y z_{xy} + (1 + z_x^2)z_{yy} = \mathcal{F}[z] = 0 \qquad & \text{in } \Omega \\ z(x, y) = g(x, y) \qquad & \text{on } \partial\Omega \end{aligned} \tag{2}$$

Clearly, this formulation satisfies the prescribed boundary values given by $g(x, y)$ due to the Dirichlet boundary condition on $\partial\Omega$. As to why the graphic of functions solving this equation describes a minimal surface, we refer to the literature. For example [1] gives a very straightforward proof.

In the following, we will call the differential operator $\mathcal{F}[\cdot]$ the Minimal-Surface Operator (MSO). The resulting partial differential equation (PDE) in eq. (2) turns out to be an elliptic PDE of second order, which is in particular *non-linear*. The solution of such a PDE is not trivial, and typically requires numerical treatment. For certain cases, analytical descriptions are available, such as for Scherk's surface.

As an example, Scherk's first surface (compare to figure 2) Σ rescaled on $\Omega = [0,1]^2$ is given by

$$\Sigma = \left\{ \left(x, y, \log\left(\frac{\cos(\pi(x-\frac{1}{2}))}{\cos(\pi(y-\frac{1}{2}))} \right) \right) \in \mathbb{R}^3 \middle| \, 0 < x, y < 1 \right\}. \tag{3}$$

which is the limit $n \to \infty$ of

$$\Sigma_n = \left\{ (x, y, u_n(x,y)) \in \mathbb{R}^3 \middle| \, 0 < x, y < 1 \right\}, \tag{4}$$

$$\lim_{y \to \pm 1} \to n \quad , 0 \leq x \leq 1 \tag{5}$$

$$\lim_{x \to \pm 1} \to -n \quad , 0 \leq y \leq 1. \tag{6}$$

The numerical solution of the MSE will require setting approbiate boundary conditions. Since $\log\left(\frac{\cos(\pi(x-\frac{1}{2}))}{\cos(\pi(y-\frac{1}{2}))}\right) \to \pm\infty$ on $\partial\Omega$, this is numerically not very practical.

We thus introduce $\beta_x, \beta_y \in (0,1)$ s.th.

$$\Sigma_{\text{trunc}} = \left\{ \left(x, y, \log\left(\frac{\cos(\pi\beta_x(x-\frac{1}{2}))}{\cos(\pi\beta_y(y-\frac{1}{2}))} \right) \right) \in \mathbb{R}^3 \middle| \, 0 < x, y < 1 \right\}. \tag{7}$$

By these means, we solve the MSE on $\Omega_{\text{trunc}} = [0,1]^2 \subset \Omega$ with $\Omega = [0, \frac{1}{\beta_x}] \times [0, \frac{1}{\beta_y}]$. Exact boundary values on $\partial\Omega_{\text{trunc}}$ ensure the correctness of the results. We choose $\beta_x = \beta_y = \beta$, as we restrict ourselves to solving on a square domain.

Later on, we will use this surface as a test-case to verify our numerical results.

2.2. Numerical solution

2.2.1. Discretization

In this project, we are supposed to solve the MSE numerically on $\Omega \equiv (0,1) \times (0,1)$. In the following, discretized quantities are indicated by a superscript h. The spatial domain is to be discretized using a structured mesh with equidistant grid spacing both in x, y, i.e. we have the same number of grid points in both directions, $N = N_x = N_y$. Thus, we define $\Omega^h := \{(x,y) \in \mathbb{R}^2 : \ (x,y) = (ih, jh),\ 0 \leq i,j < N,\ hN = 1\}$.

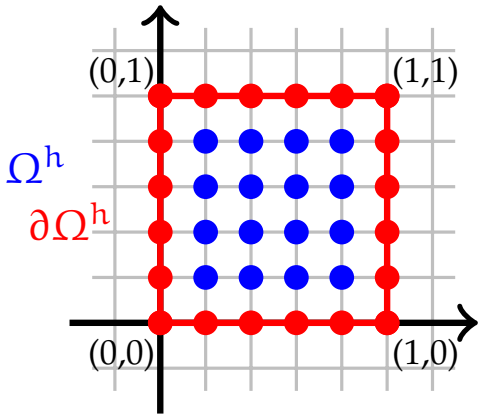

Figure 3: Depiction of Ω^h with $N = 5$, $hN = 1$. Blue: inner nodes, red: boundary nodes

We choose to discretize the MSO on Ω^h by Finite Differences, since this is usually the easiest way to go, and on a structured grid, would anyways yield similar discrete equation as in Finite Volume or Finite Element methods.

To obtain a second order consistent discrete MSO ($F^h[\cdot]$), we use central difference stencils on the first, second and mixed derivative. Since all these stencils have make only use of immediate neighours, there is no need to treat nodes close to the boundary differently, since the boundary is given by $g(\cdot)$.

This way, we obtain a discrete version of (2):

$$\begin{aligned} \left(1 + d_x[z^h]^2\right) d_{yy}[z^h] - d_x[z^h] d_y[z^h] d_{xy}[z^h] + \left(1 + d_y[z^h]^2\right) d_{xx}[z^h] = F^h\left[z^h\right] = 0 \quad &\text{in } \Omega^h \\ z^h = g \quad &\text{on } \partial\Omega^h, \end{aligned} \tag{8}$$

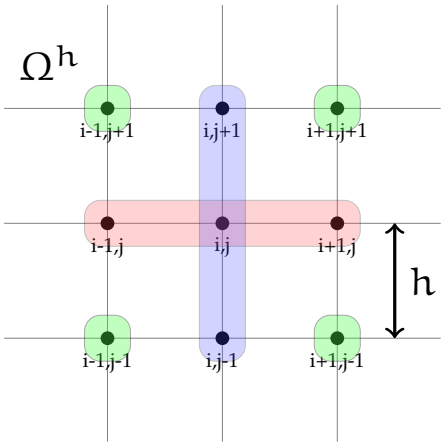

Figure 4: Snippet of the mesh with Finite Differences stencils
red/blue: first and second order stencils in x, y
green: mixed stencil

while the stencils defined on the inner nodes are given as follows ($1 \leq i, j \leq N-1$):

$$d_x[z] = \frac{z_{i+1,j} - z_{i-1,j}}{2h} \tag{9}$$

$$d_y[z] = \frac{z_{i,j+1} - z_{i,j-1}}{2h} \tag{10}$$

$$d_{xx}[z] = \frac{z_{i+1,j} - 2z_{i,j} + z_{i-1,j}}{h^2} \tag{11}$$

$$d_{yy}[z] = \frac{z_{i,j+1} - 2z_{i,j} + z_{i,j-1}}{h^2} \tag{12}$$

$$d_{xy}[z] = \frac{z_{i+1,j+1} + z_{i-1,j-1} - z_{i-1,j+1} - z_{i+1,j-1}}{4h^2}. \tag{13}$$

Figure 4 depicts the stencils on a segment of Ω^h. Since the stencils have only support to the nearest neighbors, there is no need tho treat nodes close to the boundary any different. Furthermore, this gives already rise to the conclusion, that the associated matrix will be sparse (for each of the N gridpoints, only 9 instead of N partners contribute).

2.2.2. Solution

The main difficulty in solving the MSE lies in the non-linearity of $F^h[\cdot]$. Since it is not possible, to directly invert for $z^h = \left(F^h\right)^{-1} 0$, one needs to use a procedure such as Newton-Raphson iterations. The main idea is to use some initial guess z_0^h, while in general, $F^h\left[z_0^h\right] = r_k^h \neq 0$. The goal is then to generate a sequence of z_k^h such that $r_k^h \to 0$ for increasing, but reasonably small k. Algorithm 1 shows the standard Newton-Raphson procedure, that takes as input a specific initial guess (here: 0), a tolerance for convergence TOL and the nonlinear operator $F^h[\cdot]$, outputting an approximate solution to the MSE $z_{k_{fin}}^h$.

Algorithm 1 Newton's method applied on the discrete MSE

$k \leftarrow 0$
$z_k^h \leftarrow 0$
$r_k^h \leftarrow F^h\left[z_k^h\right]$
while $\|r_k^h\| > \text{TOL}$ **do** ▷ We choose $\|\cdot\| = \|\cdot\|_2$
  $z_{k+1}^h \leftarrow z_k^h - \left(\nabla F^h\left[z_k^h\right]\right)^{-1} r_k^h$
  $k \leftarrow k + 1$
  $r_k^h \leftarrow F^h\left[z_k^h\right]$
end while

This algorithm involves the computation of the inverse of the gradient of $F^h\left[z_k^h\right]$ for each iteration k. Later, we will present and compare two different ways on how to contrive this.

It is well known, that the Newton-Raphson procedure yields fast convergence, but the success of this convergence is highly dependent on the initial guess z_0^h. Furthermore, convergence can be improved by choosing an initial guess that is closer to the solution. One possible initial guess is to take the average boundary value,
$z_{0,ave}^h = \frac{1}{|\partial\Omega^h|} \sum_{(x_i,y_j)\in\partial\Omega^h} \left(g(x_i, y_j)\right)$. While this might help convergence for surfaces, that have a certain offset with respect to the x, y-plane, it does not provide any additional information (such as preshaping the correct curvature).

To obtain a more educated initial guess, recall the MSE, equation (2). Considering only linear terms, we get

$$\begin{aligned} \mathcal{L}[z] = z_{xx} + z_{yy} = 0 \qquad & \text{in } \Omega \\ z(x, y) = g(x, y) \qquad & \text{on } \partial\Omega \end{aligned} \tag{14}$$

This linear, second order PDE is well known as the Laplace-equation, and can be solved by discretizing the second derivatives using the definitions for $d_{xx}[\cdot], d_{yy}[\cdot]$ introduced before, yielding $L[z^h] = 0$ with the usual boundary conditions. Since Laplace's equation can be regarded as a linearization of the MSE, we suspect to get a better initial guess (and thus faster and a more robust convergence behaviour) by first solving for $z_0^h = L^{-1}0$. This way, we obtain a slightly modified version of Algorithm 1, stated in Algorithm 2.

Algorithm 2 Newton's method using Laplace's Equ. as initial guess

$k \leftarrow 0$
$z_k^h \leftarrow L^{-1}0$ ▷ Solve Laplace's Equ. as initial guess
$r_k^h \leftarrow F^h\left[z_k^h\right]$
while $\|r_k^h\| > \text{TOL}$ **do**
  $z_{k+1}^h \leftarrow z_k^h - \left(\nabla F^h\left[z_k^h\right]\right)^{-1} r_k^h$
  $k \leftarrow k + 1$
  $r_k^h \leftarrow F^h\left[z_k^h\right]$
end while

2.3. Consequences for a implementation

So far we discussed the abstract numerical setting to solve the MSE. But when implementing this as a software solution, one has to consider several additional points, which especially involve the interface to the user.

We provide an overview of requirements for a potential implementation in figure 5.

We can divide communication with the user into user input and output to the user. A input interface can be realized e.g. as a GUI (graphical user interface) or as a input file (text file, that is parsed before/while running the simulations). Either realization serves the purpose to give the user a convenient possibility to set the main simulation parameters such as grid size and boundary conditions, as well as to set several tuning parameters like number of threads for parallel execution, initial guess.

The output to the user further is crucial in the sense that without giving the user any opportunity to perceive the result of the computation, there is no point in even computing anything. In terms of our application, it is required to provide the result in a form that can easily be visualized as a 3D-object. Additionally, one might want to offer a convenient validity-check and information about convergence.

Last but not least, a software implementation also needs to be able to perform all of the key steps to solve the MSE: It must allow for a discretizaion of the computational domain, and for setting the desired values on its boundary. Apart from implementing a suitable initial guess, inside a loop such as Algorithms 1, 2 there are a few important tasks missing that require special consideration: Both computing the Jacobian $\nabla \mathbf{F}^h[\cdot]$ and solving the resulting linear systems are non-trivial necessities whose treatment will be discussed in more detail in the next chapter.

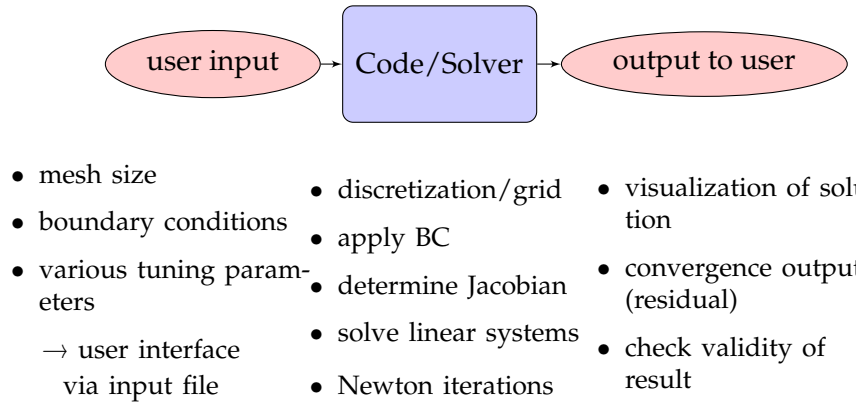

Figure 5: Requirements on a potential implementation

3. Implementation

3.1. Software/Solver Design

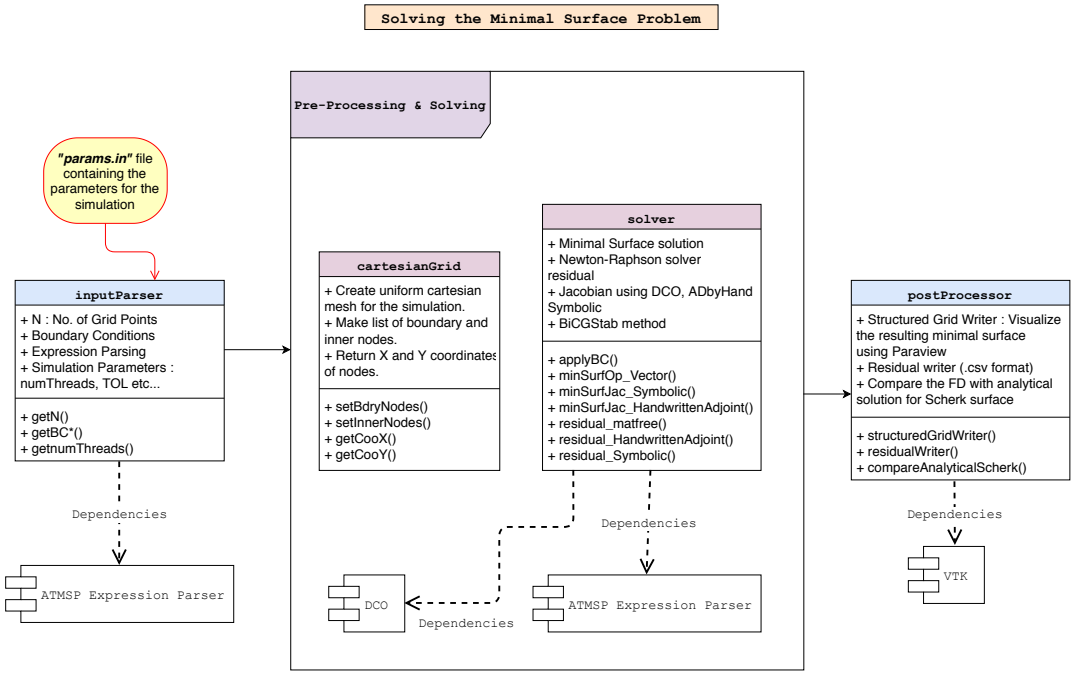

Figure 6: UML Diagram showing the strucuture of the *Minimal Surface* software.

Figure 6 shows the structure of the software, the classes and their associated functions, developed in order to solve the minimal surface problem.

- **inputParser:** This class consists of functions necessary to read all the parameters required to control and execute the minimal surface software. This class takes in `params.in` which contains all the necessary parameters such as the # of mesh elements, boundary conditions, option for the method of Jacobian computation etc... as an input to be parsed and processed. This class also depends on an external library, `ATMSP Expression Parser`, to parse the boundary conditions.

- **cartesianGrid:** This class consists of functions that help in the generation of a uniform cartesian mesh required for solving the FD problem. It also has functions required to return the `x` and `y` coordinates of a node required to apply the

boundary conditions. It also provides function calls to allocate separate storage of boundary and inner node indices.

- **solver:** This class consists of functions required to solve the minimal surface problem using Newton-Raphson method. The function to apply the boundary conditions is specified inside this class. `ATMSP Expression Parser` library is again required for this as the library provides methods to parse the boundary condition given as a string expression inside `params.in` into an executable byte code. It also uses `DCO` library to compute the Jacobian required for the Newton Raphson iterations. Separate function definitions are created to run the solver for different options of Jacobian such as Symbolic differentiation, AD by Hand and Matrix-free methods based on the option provided inside `params.in`

- **postProcessor:** This class consists of functions designed to write the solver residual as a CSV file and the resulting minimal surface as a VTS file to be visualized inside *Paraview*. To write the VTS file, the class uses the `VTK` library. The class also provides a function to compare the convergence of FD to analytical solution for the special case of a Scherk surface where the convergence rate is measured using both the *Max Norm* as well as the *l2-Norm*.

3.2. Computing the Jacobian

@Chenfei, briefly explain like in the presentation the three methods, and also name that one could use Finite Differences, but this is expected to perform bad (because...), and thus we do not use it (the next lines are already a few months old, use them or get rid of them)

Within the Newton iterations, it is necessary to compute the inverse of the Jacobian of the discrete MSO. We implemented this procedure following two different strategies:

- Based on generating the Jacobian matrix $\nabla \mathbf{F}^h \left[z_k^h\right]$ and computing its inverse
  - Hard-code a manually calculated Jacobian (to verify results)
  - Build a Jacobian matrix from Automatic Differention (AD by hand)

- Based on a linear iterative scheme, where we do not explicitly build the Jacobian, but extract its action on certain vectors from AD and feed this to a linear iterative solver

Argue that, since $\nabla \mathbf{F}$ is not symmetric in general, need stabilized CG (better performance than GMRES according to Eigen library)

Algorithm 3 Newton's method

$y \leftarrow F(z)$
while $\|y\| > \text{TOL}$ **do**
  $A = \nabla F(z)$
  $dz = A^{-1}y$ ▷ Instead of assembling the whole ∇F above, do this matrix-free
  $z \leftarrow z + dz$
  $\texttt{res} \leftarrow F(z)$
end while

Algorithm 4 Matrix-free BiCGSTAB to get dz (adapted from [2])

$z^{(1)} \leftarrow dz$
$(y, y^{(1)}) \leftarrow F^{(1)}(z, z^{(1)})$ ▷ DCO
$p \leftarrow -y - y^{(1)}$
$r \leftarrow p, r_0 \leftarrow r, \rho \leftarrow 1, \alpha \leftarrow 1, \omega \leftarrow 1$
while $\|r\| > \text{TOL}$ **do**
  $\rho_{new} = (r_0, r)$
  $z^{(1)} \leftarrow p$
  $\beta \leftarrow \rho_{new}/\rho \cdot \alpha/\omega$
  $\rho \leftarrow \rho_{new}$
  $p \leftarrow r + \beta(p - \omega v)$
  $z^{(1)} \leftarrow p$
  $(y, y^{(1)}) \leftarrow F^{(1)}(z, z^{(1)})$ ▷ DCO
  $\alpha \leftarrow \rho/(r_0, y^{(1)})$
  $dz \leftarrow dz + \alpha p$
  $s \leftarrow r - \alpha y^{(1)}$
  $z_1 \leftarrow z + dz$
  $(y_1, y_1^{(1)}) \leftarrow F^{(1)}(z_1, z_1^{(1)})$ ▷ DCO
  if $\|y_1\| < \text{TOL}$ **then**
    abort ▷ Converged at intermediate level
  end if
  $z^{(1)} \leftarrow s$
  $(y, y^{(1)}) \leftarrow F^{(1)}(z, z^{(1)})$ ▷ DCO
  $\omega \leftarrow (y^{(1)}, s)/(y^{(1)}, y^{(1)})$
  $dz \leftarrow dz + \omega s$
  $r \leftarrow s - \omega y^{(1)}$
  $z \leftarrow z + dz$
end while

3.3. Solving linear systems

I tested a bit now, and will push also to the code, timings ideally after that: seems like sparseLU is faster for getting the initial guess, so I will implement that.
for the loops, bicgstab is still advantageous as we can greatly reduce TOL_linsolver.

3.4. Testing

For testing the validity, we used Google Tests as our testing framework. The unit testing library supports multi-platforms and compilers and allows for source testing, without large source modification. Since the code was very modular, structured and independently developed in parts, unit testing was fairly simple as well. Associated data types storing computational values and Classes and their respective Functions were checked for validity. Depending on type of debugging, the library offers four broad ways of testing as represented in the figure above.

- Negation Test: In case a test fails under non-assertive statements, test function returns the expected value as compared to the manual value which makes unit debugging much simpler and faster.
- Succeed Test: Used for testing blocks or modules of code for successful compilation error. Vastly helpful for module wise checks while debugging.
- Expectational Test: States if the expected and returned value of the test matches and gives a green flag for success
- Assert Test: Fatal Testing. Used to assert program specific mandatory conditions for successful build and execution of code. Stops the compilation in case of failure.

The test part of the program gets called recursively by CMake. It creates an executable and the same can then be accessed like the main program albeit just tests with appropriate input from parameters file. All the necessary dependent header files were to be manually added for accessing the classes and their methods. For testing purposes, the code was sectioned in three broad parts as per modularity namely, Input parser, Cartesian Grid and Solver.

- Input Parser: An instance of class Input Parser (InputParserObj) is created in the testing module. Various setter functions set the values after reading the input from params file and the same can then be validated by the expected values returned by getter functions.
- Cartesian Grid: Simple unit tests pertaining to setting up of cartesian grid, sorting algorithm and the respective node lists were tested for lexicographical ordering. The same is done for both the boundary as well as inner nodes.
- Solver: Primitive solver was tested for the independent unit functional arguments. For example the functional dependency of solver for applying boundary

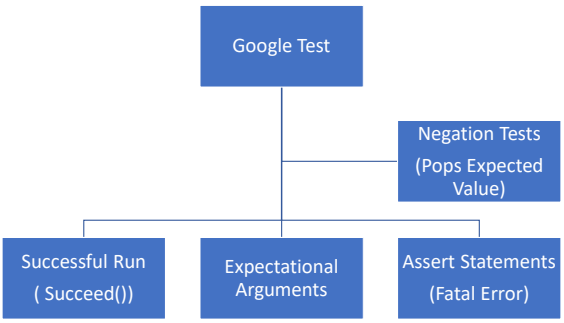

Figure 7: Testing Outputs

conditions (executed by the function applyBC()) were tested for unit arguments. A unit (or null) test vector is passed as an argument after creating an instance of the class 'solver' and calling the function by the same instance. The test vector is then tested for validity like first and last element values.

- Analytical Scherk : To test the validity of the solver, analytical solution with constrained boundary arguments are generated and error produced with respect to the solver solution. The computed error is tested for a small tolerance in maximum norm.

Remark : For testing the validity of solver against the Scherk test, the same could not be implemented in Google Tests. The issue could be narrowed down to a linker error which involves usage of the dco library as one of its dependencies and results in either 'core dump' bad memory access error failure by cmake to add tests recursively (stce error). Independent testing would demand a lot of code refactoring and hence could be put up as a scope for improvement. Hence, the same test was replicated in the post processor file and tested for a nominal tolerance.

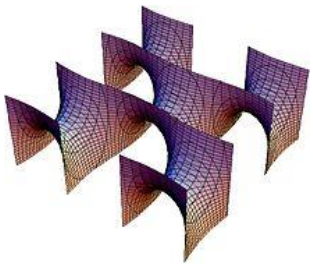

Figure 8: Nine Periods of Scherk Surface

4. Results

4.1. Scherk Surface

Scherk surface is a classic example of minimal surface. Scherk gave his two surfaces in 1834, first doubly periodic and second singly. Consider the following domain in the Euclidean plane:

$$u = (-\pi/2, +\pi/2)x(-\pi/2, +\pi/2) \rightarrow \mathbb{R}^3 \tag{15}$$

such that

$$\lim_{y \rightarrow +\underline{\pi}/2} u(x, y) = +n(-\pi/2 < x < +\pi/2) \tag{16}$$

$$\lim_{x \rightarrow +\underline{\pi}/2} u(x, y) = -n(-\pi/2 < y < +\pi/2) \tag{17}$$

Thus, u satisfies the minimal surface equation.The limiting surface as n tends to infinity is called the Scherk surface.Hence, the general Scherk surface over the square domain is given by:

$$\Sigma = \left\{ \left(x, y, \log\left(\frac{\cos(\pi(x))}{\cos(\pi(y))} \right) \right) \in \mathbb{R}^3 \middle| + \pi/2 < x, y < -\pi/2 \right\}. \tag{18}$$

4.2. Timings

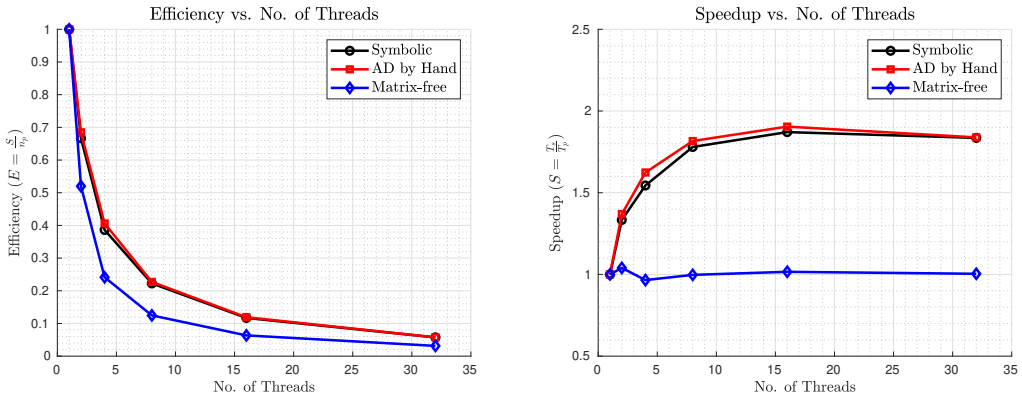

Figure 9: Parallel Efficiency and Speedup plots for different Jacobian methods

The code was timed for effectiveness of parallelization. It can be seen that for N = 1000, both Symbolic and AD by Hand take up as much time as Matrix-free method for N = 500. Although the Matrix-free method takes more time per iteration it requires fewer iterations to converge as when compared to the other two methods. This can be asserted mainly due the fact that every dco iteration has multiple Newton iterations which essentially makes the process much slower. Also, there is no due criteria to check for symmetricity.

As an example set, for N=500 , Newton's tolerance as 10^{-6} and linear solver tolerance as 10^{-7} , we get the time for the execution of the code as 279.220 [s] on average and when we use Newton's tolerance as 10^{-3} and linear solver tolerance as 10^{-7}, we get the execution time as 155.364 [s] which is half the time compared to the previous case. For both these cases, we get the same apriori error measured in both l2-norm and max norm.

5. Conclusions and outlook

- Code could be modified to better suit for testing purposes. Some test methods were not ethically called due to private dependencies. Appropriate return functions would make it easier and simpler
- Private methods could be tested by yet another way: inclusion of `Gtest_prod.h` . Unfortunately, not supported by standard CMake.

- Extension of the minimal surface problem to include time varying boundary conditions to account for evolution of solution w.r.t. time
- Learnt about digital collaboration per se. Developing a project from the scratch. Had all the different applications from Simulation Sciences.

References

[1] Makoto Sakai. *Lecture Notes in Mathematics – Minimal Surfaces in R3*. 1976.

[2] Uwe Naumann. *The art of differentiating computer programs: an introduction to algorithmic differentiation*, volume 24. Siam, 2012.

A. User manual

This section provides a short description on how to run the code. The software accompanying this report is self-contained and guaranteed to run on the RWTH cluster, using either the current versions of the Intel (19) or GCC (9) compiler.

A.1. Compilation

To compile the code, open a terminal inside `source/`.

```
cd build
cmake ..
make
```

The standard setup is the CMAKE compile option `Release` admitting some compiler optimization. If some additional debug information is desired, change the option to `debug` inside `CMakeLists.txt`.

A.2. Running a simulation

To run a simulation, go to the folder testcases or create a new folder with a valid `params.in`-file.

Inside this `params.in`-file, one can find the options to set with descriptions.
To run the a simulation, execute the following, once the input file is set up:

```
cd <folder with params.in>
export NAG_KUSARI_FILE=<path to nag_key.txt>
   # needed if jacobianOpt=2
ln -s <path to executable minSurf in build/> .
   # symbolic link for convenient execution
./minSurf
```

Now, the resulting output can be inspected by opening the `.vts`-files inside `surfaces/` using Paraview[1]. If one is interesting in the convergence, one can find the residual data over iteration inside `residual/`, which can be also plotted using Paraview.

[1] to use this on the cluster, execute `module load GRAPHICS paraview` beforehand.