Report
Simulation Sciences Laboratory

Minimal Surfaces

Chenfei Fan 390189
Praveen Mishra 389850
Sankarasubramanian Ragunathan 389851
Philipp Schleich 391779

February 28, 2020

Supervisor: Prof. Dr. Uwe Naumann
Klaus Leppkes

Contents

1. Introduction

This report concludes our SiSc Laboratory Project in WS19/20, with the goal, to implement a software, that is able to numerically compute so called *Minimal Surfaces*. As one might intuitively think, when we speak about surfaces, minimal has something to do with its surface. To this end, we define a *Minimal Surface* to be a surface that locally minimizes its area (that is, the surface has a zero mean curvature over its domain). A trivial example is given by a plane. [1]

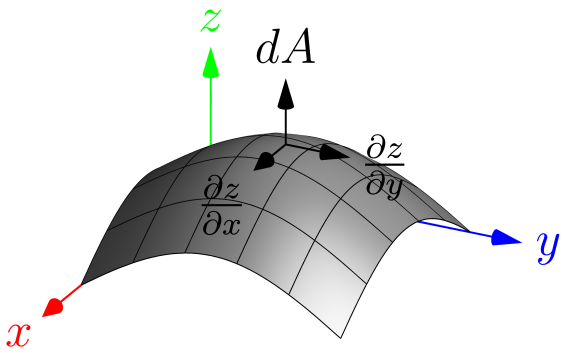
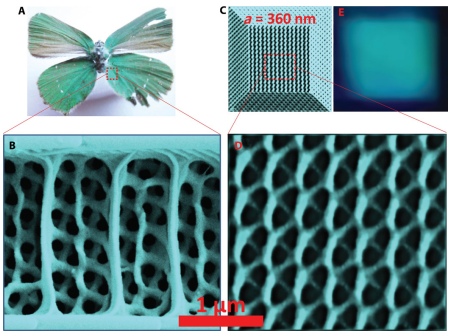


Figure 1: A surface with the local area vector.
For this surface to be minimal, $\|\mathrm{d}\mathbf{A}\| \to \min$

Minimal surfaces appear in nature as well as they have some applications in engineering, while they can be observed both in macro- and microscopic length scale.

- Macroscopic scale:
  - Architectural design: Designing structures as minimal surfaces
  - Soap bubbles and dew drops
  - Relativistic description of black holes
- Microscopic scale:
  - Material lattice structure
  - Hydrophobic co-polymer structures
  - Butterfly wings

Structures whose geometry makes up a minimal surface typically have large tensile strength, which can be used e.g. for man-made lightweight design in civil engineering (cf. figure 2a) or lightweight design given by evolution, such as butterfly wings (gyroids, cf. figure 2b. MSE is also used to provide a relativistic description on the



(a) [2, Gyroid minimal surface]

(b) The West German pavilion, designed by Frei Otto.[3]

formation of Black Holes. Material lattice structures of biological organisms such as butterfly wings are minimal surface structures). 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. [2, 1]

2. Background

2.1. Mathematical basics

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 $\partial\Omega$. [4]

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 two-dimensional domain, particularly in a subset of $\mathbb{R}^2$. This function z has to fulfill the so called *Minimal-Surface Equation* (MSE) stated below.

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

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 [4] 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. (1) 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.

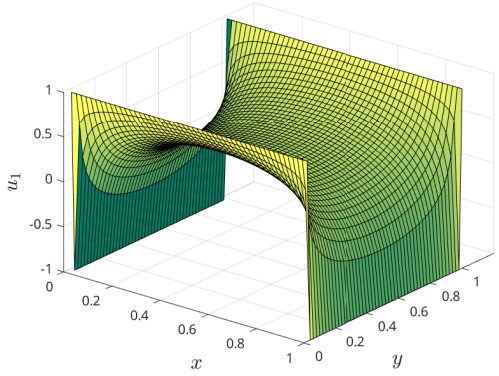


Figure 3: Scherk's first surface on $[0,1]^2$

As an example, Scherk's first surface (compare to figure 3) Σ rescaled on $\Omega = [0,1]^2$ is given by

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

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

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

$$\lim_{y \to \pm 1} \to n \quad , 0 \leq x \leq 1 \tag{4}$$

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

[5]

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.

$$\Sigma_{\text{trunc}} = \left\{ \left(x, y, \log\left(\frac{\cos(\pi\beta_x(x-\frac{1}{2}))}{\cos(\pi\beta_y(y-\frac{1}{2}))} \right) \right) \in \mathbb{R}^3 \middle| \, 0 < x, y < 1 \right\}. \tag{6}$$

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 $\Omega^h := \{(x,y) \in \mathbb{R}^2 : \ (x,y) = (ih, jh),\ 0 \leq i,j < N,\ hN = 1\}$.

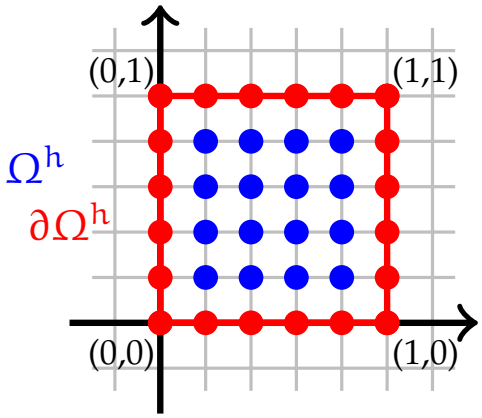


Figure 4: Depiction of Ω^h with $N = 5$, $hN = 1$. Blue: inner nodes, red: boundary nodes

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, assuming that the underlying solution is at least twice continuously differentiable. This can be perceived as a constraint on the boundary conditions that are set in the numerical solution: In case g does not have sufficient regularity, convergence is not guaranteed. 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 $g(\cdot)$.

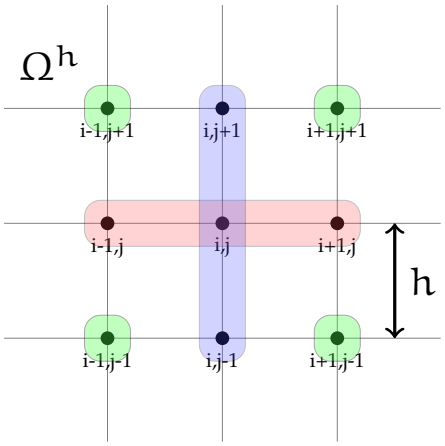


Figure 5: Snippet of the mesh with Finite Differences stencils
red/blue: first and second order stencils in x, y
green: mixed stencil

This way, we obtain a discrete version of (1):

$$\left(1 + d_x[z^h]^2\right) d_{yy}[z^h] - d_x[z^h] d_y[z^h] d_{xy}[z^h] + \left(1 + d_y[z^h]^2\right) d_{xx}[z^h] = F^h\left[z^h\right] = 0 \quad \text{in } \Omega^h$$
$$z^h = g \quad \text{on } \partial\Omega^h, \tag{7}$$

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

$$d_x[z]_{i,j} = \frac{z_{i+1,j} - z_{i-1,j}}{2h} \tag{8}$$
$$d_y[z]_{i,j} = \frac{z_{i,j+1} - z_{i,j-1}}{2h} \tag{9}$$
$$d_{xx}[z]_{i,j} = \frac{z_{i+1,j} - 2z_{i,j} + z_{i-1,j}}{h^2} \tag{10}$$
$$d_{yy}[z]_{i,j} = \frac{z_{i,j+1} - 2z_{i,j} + z_{i,j-1}}{h^2} \tag{11}$$
$$d_{xy}[z]_{i,j} = \frac{z_{i+1,j+1} + z_{i-1,j-1} - z_{i-1,j+1} - z_{i+1,j-1}}{4h^2}. \tag{12}$$

Figure 5 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 (1). Considering only linear terms, we get

$$\begin{aligned} \mathcal{L}[z] = z_{xx} + z_{yy} = 0 \qquad & \text{in } \Omega \\ z(x,y) = g(x,y) \qquad & \text{on } \partial\Omega \end{aligned} \tag{13}$$

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 6.

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.

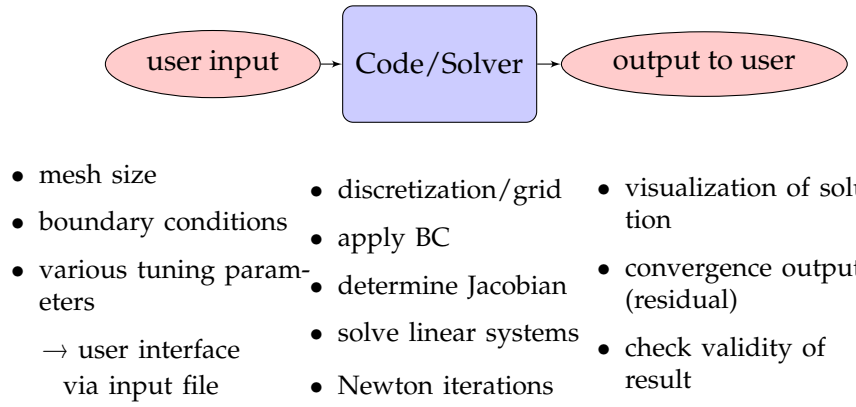


Figure 6: Requirements on a potential implementation

3. Implementation

3.1. Software/Solver Design

In order to meet the requirements, that have been identified earlier, we create four different classes. These classes are individually responsible for a certain subset of tasks.

A user interface will be given by a input file, called `params.in`, which is parsed by the class `inputparser`. Creating an object of `inputParser` thus allows us to access the input file from any other place in the code. Further, `cartesianGrid` takes care of the discretization, and an instance of such a grid enables to access the specifics of our equidistant, square mesh. The most important part of our code is made up by the class `solver`, which contains all required numerical (sub-)routines. Last but not least, the `postProcessor`-class builds the connection to the user after having completed a simulation, providing information about solution and convergence.

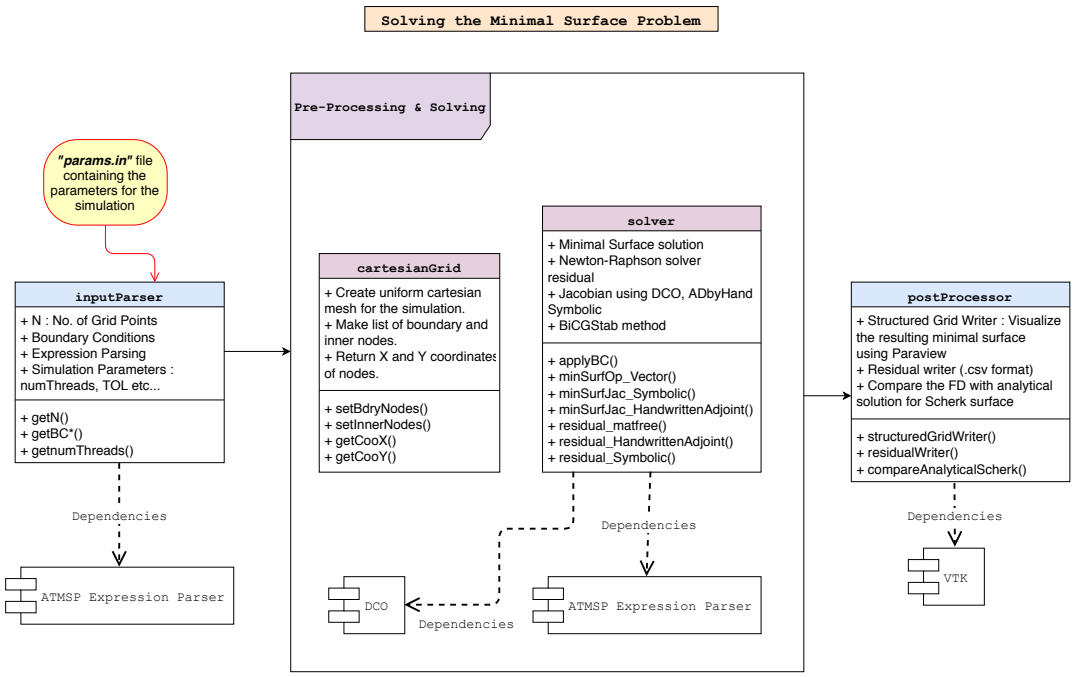


Figure 7: UML Diagram with the strucuture of the *minSurf* software.

Figure 7 shows the structure of the software, the classes and their associated functions, developed in order to solve the minimal surface problem. In the following, the individual classes are again described in more detail.

- `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 number of mesh elements[1], 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`[2], 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 Finite Difference discretization. It also has functions able to return the x and y coordinates of a node required to apply the boundary conditions. Moreover, it provides methods 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 contains functions designed to write the solver residual as a CSV file and the resulting minimal surface as a VTS file allowing 3D visualization using *Paraview*. To write the VTS file, the class uses the `VTK` library. It also provides a function to compare the convergence of the numerical towards the analytical solution for the special case of the Scherk surface. The error is measured using both the maximum-norm ($\|z - z^h\|_\infty := \max\limits_{1 \le i,j \le N} |z_{ij} - z^h_{ij}|$) as well as the ℓ_2-norm ($\|z - z^h\|_2 := \sqrt{\frac{1}{N^2} \sum_{i,j=1}^{N} \left(z_{ij} - z^h_{ij}\right)^2}$).

[1] represented by N, the number of elements per dimension

[2] We modified this parser slightly, such that it is also able to parse piece-wise boundary conditions. The original library is available at https://sourceforge.net/projects/atmsp/

3.2. A note on solving linear systems (does not affect the matrix-free solver)

When solving the MSE, we deal in general with sparse, non-symmetric systems. The `Eigen` library (we use the release Eigen 3.3.7) provides the following sparse linear algebra methods[3], that would be suitable for our problem:

- SparseLU – a sparse version of LU-decomposition
- BiCGSTAB – Stabilized biconjugate gradient descent (more costly than CG, but suitable for general square matrices). Essentially, this represents a CG-routine inside GMRES(1)
- GMRES – Generalized Minimum Residual Method

The latter two are linear iterative schemes suitable for general matrices, while we exclude GMRES from now on, as BiCGSTAB is slightly more efficient according to benchmarks of the `Eigen` library[4]. In contrast to LU decomposition, which is a direct method, linear iterative schemes might be faster, but do not guarantee convergence. This depends on the system to be solved. As it turns out, the MSE using Finite Differences yields sufficiently well behaved systems, that typically converge.

The convergence of iterative schemes can be (dependent on the system) boosted by the use of preconditioners. In `Eigen`, either the identity, a Jacobi (diagonal, iff non-zero) or a incomplete LU-decomposition (ILUT) preconditioner are available, while the latter in general has the most potential to speed up convergence.

Testing these methods on our initial-guess subroutine, which solves the Laplace-equation, has yielded, that, while SparseLU is faster than BiCGSTAB+Jacobi, the right tuning of ILUT in combination with BiCGSTAB is fastest.

Still though, our implementation inside the Newton-iteration uses bare BiCGSTAB with Jacobi-preconditioner. The reason therefore is, that we chose to represent the discrete system not as the reduced system (i.e. boundary information is brought onto the right-hand-side, yielding a system size of $(N-2)^2$ instead of N^2), but as the full system[5]. While this has no impact on the feasibility of a LU-decomposition when solving the Laplace-equation, it prohibits a LU-decomposition on the full system.

[3] Due to the severe sparsity of the matrix, we did not consider a dense matrix implementation.

[4] `http://eigen.tuxfamily.org/dox/group__TopicSparseSystems.html`, in the very bottom, assuming both use the ILUT preconditioner

[5] We discussed this with our supervisor in the beginning of the project, and concluded that solving the full system would not have substantial drawbacks and would be more intuitive to implement.

Solving the Laplace-equation can be done in one step – this means, we prescribe the boundary information by putting the associated values on the right-hand-side, and insert an (almost) empty row[6], that carries a 1 on the place, that is to be multiplied with the boundary value at its respective position. In conclusion, the system matrix, that is to be inverted, has still full rank $\Leftrightarrow$ does not contain an empty row. What changes now for the linear systems inside the Newton iterations? We start with an initial guess $z^{(0)}$, which already must contain the boundary information. The linear systems arise, when we want to get from $z^{(i)}$ to $z^{(i+1)}$ – the difference $dz^{(i)}$ resulting from the linear solve must not change the boundary information, since both $z^{(i)}$ and $z^{(i+1)}$ inherit already the correct values from $z^{(0)}$. As a consequence the Jacobian of the MSO $\nabla F^h\left[z^{(i)}\right]$ has an empty row, whenever the row index corresponds to a boundary node. This would prevent us from using SparseLU or an iterative scheme with the ILUT preconditioner. BiCGSTAB+Identity still works, since it only relies on matrix-vector products and scalar products, and multiplying empty rows is not forbidden (but unnecessary work)[7].

We can overcome this by putting some little extra effort, and setting a 1-entry on the diagonal of the Jacobian of the discrete MSO, if the row corresponds ot a boundary index. This does not change the result, since we are exact on the boundary for all iterations, thus the right-hand-side in this case is 0. By such a procedure, we end up with a non-singular Jacobian, and can use direct methods and preconditioners. As before, numerical tests suggest that for the right tuning, BiCGSTAB+ILUT is the fastest option for our problem, while SparseLU would be still faster that BiCGSTAB+Jacobi. Next, note, that the Newton procedure does not require the 'exact' value of $dz^{(i)}$, but can be satisfied with a solution up to a certain tolerance[8], on one hand enables us to choose this tolerance of the same order[9]. This allows us to additionally boost the runtime of the iterations by choosing a smart value for this linear solve tolerance.

Therefore, we choose this to be the default in our solver. We are not providing a switch option to change the solver, because the software is only capable of solving this very PDE, which means we do not have to expect a substantially different structure of the Jacobian[10]. Apart from that, as stated before, the iterative scheme does not rely on exact updates, which can only be exploited using iterative schemes. Therefore, a single, but optimal, solver option shall be sufficient.

[6] With empty row, we mean a row, that is only made up by 0-elements.
[7] Jacobi preconditioner also can be used, since it only acts if the diagonal entry is non-zero
[8] keyword in `params.in`: $\mathrm{TOL}_{\mathrm{linsolver}}$
[9] in some cases even slightly larger – this requires possibly more iterations, while each linear solve is less and less costly
[10] Depending on boundary conditions, it has different entries, but the non-zero pattern stays the same in general.

3.3. Computing the Jacobian

Within the Newton iterations, it is necessary to compute the inverse of the Jacobian of the discrete MSO and solve the resulting linear systems. One well-known approach is using Finite Differences to further discretise the discrete MSO, but it is not efficient when the mesh is very fine. As a result, Algorithmic Differentiation is used in this problem. We implemented this procedure following two different strategies:

- Build the Jacobian by Algorithmic Differentiation (AD by hand) and compute its inverse by existing linear solvers.
- Do not explicitly build the Jacobian, but use existing AD libraries to extract its action on certain vectors. Substitute this into the self-implemented matrix-free linear solver.

To get an overview of the problem's dimensions and properties as well as to provide a solution to verify other approaches based on Algorithmic Differentiation, we first implemented a straightforward ("hard-coded") method to derive the required Jacobian $\nabla F(z)$ by substituting Equ. (8)-(12) into Equ. (7) and taking its derivatives of z. As is indicated in section 2.2.1, the resulting Jacobian is sparse, so it is stored in `Eigen::SparseMatrix` format by `Eigen` library. Then solve $dz = \nabla F^h\left[z_k^h\right]^{-1} y$ by Bi-conjugate gradient stabilized method (BiCGSTAB). It is used instead of BiCG or CG since $\nabla F^h\left[z_k^h\right]$ is not symmetric in general.

3.3.1. AD by hand

As is shown in section 2.2.1, the discrete MSO of each gridpoint has only support from its 9 neighbors. Considering the derivation of the Jacobian as a loop of gridpoint-wise MSO over each gridpoint (which is parallelizable in nature), the gridpoint-wise discrete MSO for each gridpoint (i,j) can be generalized as Equ. (14), where h is a function which has 1 output $F(z[i][j])$ and 9 inputs $z[i'][j'], i' \in \{i, i\pm1\}, j' \in \{j, j\pm1\}$. Given this function primal of $F : \mathbb{R}^n \to 1$ (here $n = 9$), it is advantageous to use a first-order adjoint model as Equ. (15), (16). By seeding the output $y_{(1)}$ one can harvest the whole Jacobian row-wise, where one row of the Jacobian has only 9 non-zero entries. The function signature of the overloaded function is stated in Equ. (17). By the driver function in Equ. (18) (assume an inner gridpoint (i,j) has index idx), one column of the Jacobian $a1\_z[idx][:]$ together with the primal $z[idx]$ is harvested for each gridpoint (i,j).

$$F(z[i][j]) = h\left(z[i'][j'], i' \in \{i, i\pm1\}, j' \in \{j, j\pm1\}\right) \tag{14}$$

$$y := F(\mathbf{x}) \tag{15}$$

$$\mathbf{x}_{(1)} := \nabla F(\mathbf{x})^{\mathsf{T}} \cdot y_{(1)} \tag{16}$$

$$\left(y, \mathbf{x}_{(1)}\right) := F_{(1)}\left(\mathbf{x}, y_{(1)}\right) \qquad F_{(1)} : (\mathbb{R}^n \times 1) \to (1 \times \mathbb{R}^n) \tag{17}$$

$$(z[\mathrm{idx}],\ a1\_z[\mathrm{idx}][:]) = F\,(z[:],\ a1\_Fz[\mathrm{idx}]) \tag{18}$$

The row-wise derivation of the Jacobian inside a loop over all inner gridpoints can be decomposed into a single assignment code (SAC). Example of code snippets for each inner gridpoint if index i are as Code 1, 2. Because of the sparsity of the Jacobian, the computational complexity for each inner gridpoint is a constant. Therefore, the total computational complexity is $\mathcal{O}(N^2)$ for N^2 gridpoints (or to say, N^2 rows of the Jacobian).

Same as the "hard-coded" method, we store the derived Jacobian in `Eigen::SparseMatrix` format and solve the linear system by BiCGSTAB.

Results have proven that we apply Algorithmic Differentiation to the problem and get a feasible result. However, since it is "by hand", it has certain limitations. If the problem is more complex, it is inefficient, or even impossible to manually derive it step by step. A more general approach is still needed to automatically derive the Jacobian without manual computation.

Listing 1: Adjoint SAC Code for inner gridpoint i (primal/forward)

```
// d(...) = fd(x)
dx = (z[i+1]-z[i-1])/(2*h);
dy = (z[i+N]-z[i-N])/(2*h);
dxx, dxy, dyy = ...

// intermidiates v1, v2, v3
v1 = (1+pow(dx,2))*dyy;
v2 = -2*dx*dy*dxy;
v3 = (1+pow(dy,2))*dxx;

// result
Fz[i] = v1+v2+v3;
```

Listing 2: Adjoint SAC Code for inner gridpoint i (adjoint/reverse)

```
(initialize everything to zero)
// seed a1_Fz
a1_Fz[i] = 1.0;

// result Fz[i] = v1+v2+v3
a1_v1 = a1_Fz[i];
a1_v2 = a1_Fz[i];
a1_v3 = a1_Fz[i];

// intermidiates v1, v2, v3
a1_dy = dxx*2*dy*a1_v3;
a1_dxx = (1+pow(dy,2))*a1_v3;
...

// d(...) = fd(x)
a1_z[i][i+1] = a1_dx/(2*h)+...;
a1_z[i][i-1] = a1_dx/(-2*h)+...;
...
```

3.3.2. Matrix-free linear solver with `dco_c++`

`dco_c++` is a library that implements Algorithmic Differentiation by overloading in C++. It provides the same function as AD by hand in terms of harvesting the Jacobian. However, if all manual computation is eliminated, it is unable to exploit the sparsity of the Jacobian, so we need $\mathcal{O}(N^4)$ space to store it, which for large N is impossible. Therefore, we implemented a matrix-free linear solver inside Newton's method as described in Algorithm 3, which does not require explicitly building the Jacobian, but instead apply the vector tangent code overloaded by `dco_c++` as a matrix-vector multiplication tool which multiplies the Jacobian with the seed to harvest the result.

In general, the MSO is an operator of $F : \mathbb{R}^{N\times N} \to \mathbb{R}^{N\times N}$, which has the same computational complexity for vector tangent mode and vectors adjoint mode. We choose vector tangent mode in our implementation.

Algorithm 3 Newton's method

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

The first-order vector tangent mode for $\mathtt{F} : \mathbb{R}^{N\times N} \to \mathbb{R}^{N\times N}$ is defined as Equ. (19), (20), and its function signature is defined in Equ. (21). By substituting Equ. (19) into an existing linear solver (for example BiCGSTAB, as we implemented in Algorithm 4), $\mathbf{dz} = \mathbf{A}^{-1}\mathbf{y}$ is solved by this iterative approach.

The advantage of the matrix-free linear solver with `dco_c++` is that it does not require any prior knowledge of the operator. Given any primal `y = F(z)` where $\mathtt{F} : \mathbb{R}^{N\times N} \to \mathbb{R}^{N\times N}$, only several lines of code is needed for the vector tangent mode by overloading, as is shown in Code 3.

Listing 3: Vectors of Tangents by Overloading

```
// activate
std::vector<ADtype> z_(std::begin(z),std::end(z));
// seed
for (auto& i: grid.innerNodeList)
dco::derivative(z_)[i] = dz[i];
// compute
y_ = F(z_); // overloaded
// harvest
dy = dco::derivative(y_);
```

$$\mathbf{y} := \mathtt{F}(\mathbf{x}) \tag{19}$$

$$Y^{(1)} := \nabla \mathtt{F}(\mathbf{x}) \cdot X^{(1)} \tag{20}$$

$$\left(\mathbf{y}, Y^{(1)}\right) := \mathtt{F}^{(1)}\left(\mathbf{x}, X^{(1)}\right) \qquad \mathtt{F}^{(1)} : \left(\mathbb{R}^{N\times N} \times \mathbb{R}^{N\times N}\right) \to \left(\mathbb{R}^{N\times N} \times \mathbb{R}^{N\times N}\right) \tag{21}$$

Algorithm 4 Matrix-free BiCGSTAB to get $\mathbf{dz}$ (adapted from [6])

$z^{(1)} \leftarrow \mathbf{dz}$
$(\mathbf{y}, \mathbf{y}^{(1)}) \leftarrow \mathtt{F}^{(1)}(z, z^{(1)})$ ▷ DCO
$\mathbf{p} \leftarrow -\mathbf{y} - \mathbf{y}^{(1)}$
$\mathbf{r} \leftarrow \mathbf{p}, \mathbf{r}_0 \leftarrow \mathbf{r}, \mathbf{p} \leftarrow 0, \boldsymbol{\nu} \leftarrow 0, \rho \leftarrow 1, \alpha \leftarrow 1, \omega \leftarrow 1$
while $\|\mathbf{r}\| > \mathrm{TOL}$ **do**
  $\rho_{\mathrm{new}} = (\mathbf{r}_0, \mathbf{r})$
  $\beta \leftarrow \rho_{\mathrm{new}} / \rho \cdot \alpha / \omega$
  $\rho \leftarrow \rho_{\mathrm{new}}$
  $\mathbf{p} \leftarrow \mathbf{r} + \beta(\mathbf{p} - \omega\boldsymbol{\nu})$
  $z^{(1)} \leftarrow \mathbf{p}$
  $(\mathbf{y}, \mathbf{y}^{(1)}) \leftarrow \mathtt{F}^{(1)}(z, z^{(1)})$ ▷ DCO
  $\boldsymbol{\nu} \leftarrow \mathbf{y}^{(1)}$
  $\alpha \leftarrow \rho / (\mathbf{r}_0, \boldsymbol{\nu})$
  $\mathbf{dz} \leftarrow \mathbf{dz} + \alpha\mathbf{p}$
  if $\|\mathtt{F}(z + \mathbf{dz})\| < \mathrm{TOL}$ **then**
    abort ▷ Converged at intermediate level
  end if
  $\mathbf{s} \leftarrow \mathbf{r} - \alpha\boldsymbol{\nu}$
  $z^{(1)} \leftarrow \mathbf{s}$
  $(\mathbf{y}, \mathbf{y}^{(1)}) \leftarrow \mathtt{F}^{(1)}(z, z^{(1)})$ ▷ DCO
  $\omega \leftarrow (\mathbf{y}^{(1)}, \mathbf{s}) / (\mathbf{y}^{(1)}, \mathbf{y}^{(1)})$
  $\mathbf{dz} \leftarrow \mathbf{dz} + \omega\mathbf{s}$
  $\mathbf{r} \leftarrow \mathbf{s} - \omega\mathbf{y}^{(1)}$
end while
$z \leftarrow z + \mathbf{dz}$

There are also drawbacks of the matrix-free linear solver. One is already noted, that it is not as flexible as AD by hand to exploit the structure of the Jacobian (for example, sparsity) to reduce computation and space complexity. It remains an interesting topic for researches. Another issue is that the mostly sequential matrix-free solver is not easy to parallelize. Although there exist some better implementations than ours, it is not user-friendly to require any implementation of linear solvers. For example, if the user knows the Jacobian is symmetric, a CG method which is faster than BiCGSTAB must be implemented from scratch.

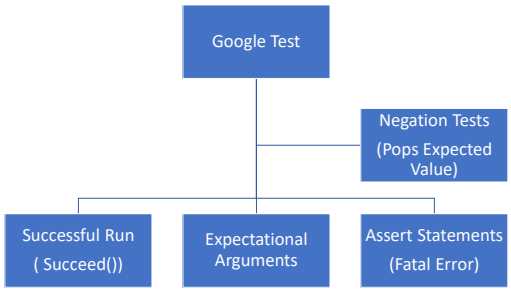


Figure 8: Testing functionalities offered by Google Tests

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 methods were checked for validity. Depending on the 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. Very 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 in case of success.
- Assert Test (Fatal Testing): Used to assert program specific mandatory conditions for a 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 the `params.in` file. All the necessary dependent header files had 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, `inputParser`, `cartesianGrid` and `solver`.

- `inputParser`: 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.

- `cartesianGrid`: 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 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: Testing the validity of solver against the Scherk test 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 `postprocessor` file and tested for a nominal tolerance, chosen to be β^2. This choice is motivated by the fact, that for smaller β, the solution is better behaved and the resulting error shall be smaller. As an upper bound, we want the error never to be larger than 1.

4. Results

In the following, we will show screenshots of three different test-cases.

4.1. Scherk Surface

Recall the formulation introduced in section 2.1

$$\Sigma_{\text{trunc}} = \left\{ \left(x, y, \log\left(\frac{\cos(\pi\beta_x(x-\frac{1}{2}))}{\cos(\pi\beta_y(y-\frac{1}{2}))} \right) \right) \in \mathbb{R}^3 \,\middle|\, 0 < x, y < 1 \right\}. \tag{22}$$

where we introduced a scaling factor β[11], which scales the boundaries and effectuates solving the MSE on a truncated domain. We can use the above expression then to compute the error of the numerical approximation with respect to a analytical solution and thereby validate the correctness of our code.

For now $\beta_1 = 0.4$, $\beta_2 = 0.8$, we see the result in figure 9.

(a) $\beta_1 = 0.4$ (b) $\beta_2 = 0.4$

Figure 9: Scherk's surface on a truncated domain for different choices of the scaling factor β

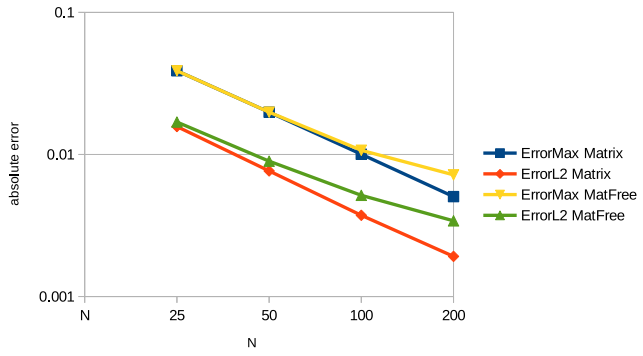
Note, that our method converges for $\beta \lessapprox 0.85$, depending on N and the Jacobian. As we can see in figure 9, the surfaces corresponding to larger values of β are less well behaved, thus typically show larger error values with respect to the exact solution. To this end, we look at a brief convergence study for $\beta_1 = 0.4$, $\beta_2 = 0.8$, using the matrix-free Jacobian[12].

Testing with a choice of options for the Jacobian and for β gives us the following insights (cf. figure 10):

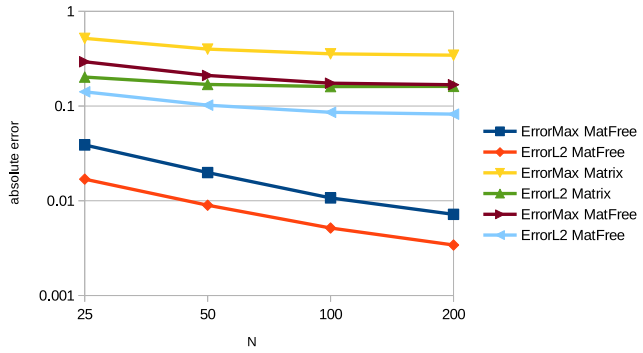
[11] keyword/constant in `params.in`: $SCALE

[12] The other Jacobians yield comparable results, and are not depicted here for the sake of a better overview

- The matrix-based algorithms and the matrix-free algorithm do not seem to be totally equivalent for large β and/or for large number or gridpoints and small β. While they both converge to a solution with acceptable error , and the plot of the solution cannot be distinguished by eye for larger β, the final error for the matrix-based algorithm tends to be a little lower.
- On the other hand, the matrix based algorithms appear to be less suitable for $\beta \to 1$.
- Further, we can observe, that for $\beta \gtrapprox 0.85$, the matrix based algorithm does still converge, but the matrix-free method does not (even with a small relaxation parameter).



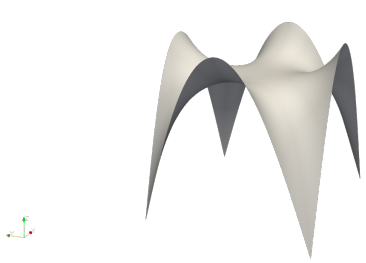
(a) $\beta_1 = 0.4$



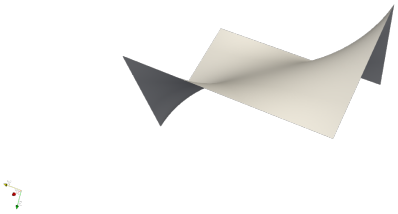
(b) $\beta_2 = 0.4$

Figure 10: Scherk's surface on a truncated domain for different choices of the scaling factor β

From figure 10 we can deduce, that for small enough β and thus well enough behaved boundary functions, we get roughly first order of convergence in the error.



(a) Sine-testcase



(b) Piece-wise boundary conditions introduced in section 4.3. The boundary `bottom` is here on the left side, `top` being on the right.

Figure 11: Visualization of surfaces obtained by a specific choice piece-wise boundary conditions

4.2. Sine boundaries

The second testcase implemented uses $g|_{\text{bottom}} = g|_{\text{top}} = \sin(\pi(x - 0.5))$ and $g|_{\text{left}} = g|_{\text{right}} = \sin(\pi(y - 0.5))$. By running a simulation with standard parameters (standard `params.in`-file), we get the surface in figure 11a.

4.3. Piece-wise boundary conditions

Furthermore, our `inputParser` using a modified ATMSP-library is able to parse piece-wise defined boundary conditions. To contrive this, we implemented a kind of indicator functions, defined as $\mathsf{smaller1}(x) : \mathbb{R} \to \{0, 1\}, \mathsf{larger1}(x) : \mathbb{R} \to \{0, 1\}$

$$\mathsf{smaller1}(x) = \begin{cases} 1 & \text{, if } x \leq 1 \\ 0 & \text{, else} \end{cases}, \quad \mathsf{larger1}(x) = \begin{cases} 1 & \text{, if } x \geq 1 \\ 0 & \text{, else} \end{cases} \tag{23}$$

Using these two functions, while scaling x on the interval one wishes to constrain on, it is able to parse piece-wise defined functions.

As an example, we chose the following: On the left and right boundary, $g|_{\text{left}} = g|_{\text{right}} = 0$, while on the upper (top) boundary, we choose a two lines, meeting continuously in the middle above 0, on the lower (bottom) boundary, they are mirrored along the x, y plane, and meet below 0 (this is not smooth, but will also yield a valid surface).

The result can be found in figure 11b.

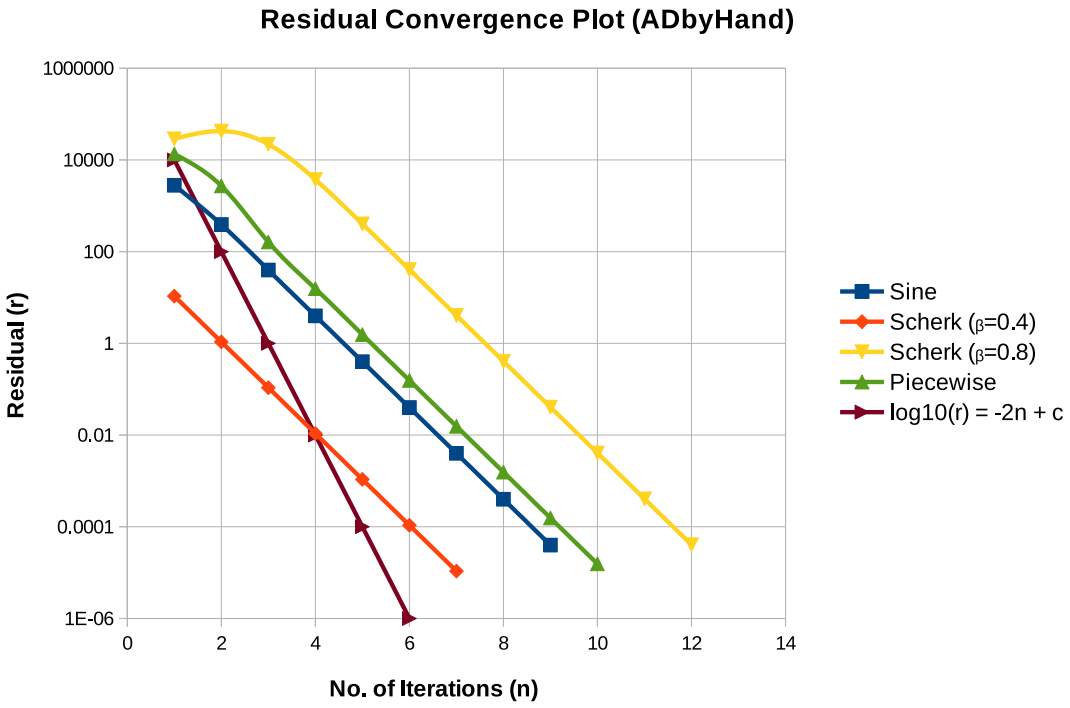
4.4. Interpretation of convergence results

At last, we investigate the convergence behaviour of the different Jacobians implemented. Since adjoint AD by hand and symbolic differentiation basically correspond to each other, we only depict the former here.

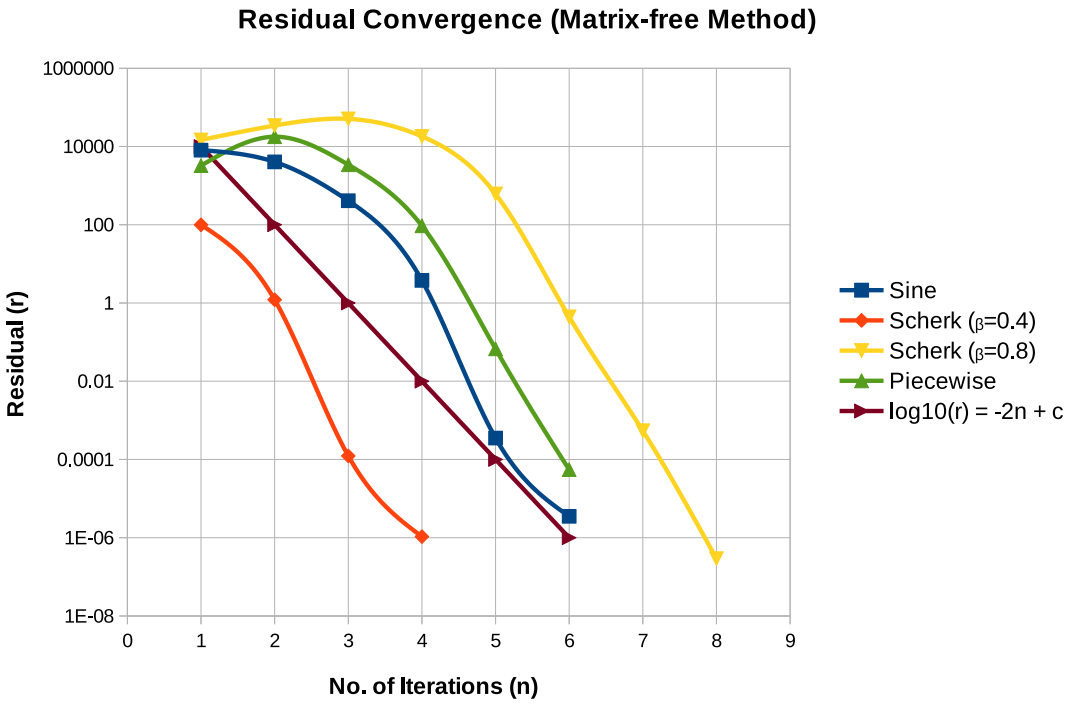
It can be easily seen, that the matrix-free algorithm needs much less Newton iterations to converge. The reason therefore lies in the fact, that we cannot directly compare one matrix-free Newton iteration with one iteration of the algorithms assembling a matrix. The latter linearize the MSE once per Newton iteration, and then solve for the update step. The former basically builds a new linearization within each step of the underlying linear systems algorithms, since the action of the Jacobian is always considered on the updated solution. This results in less iterations needed, while considering the run-time (section 4.5) While the matrix-free method converge approximately in second order, the matrix-assembling methods are approximately first order. Also, one can see, that the matrix-free convergence follows a non-linear curve, in contrast to using a matrix-depending algorithm.

Apart from that, we observe, that for both methods, the hierarchy of the convergence per testcase ($\leftrightarrow$ boundary condition) is the same. The slowest convergence is obtained by Scherk's surface with a large scaling factor $\beta = 0.8$, which is expected from our findings in section 4.1. The nicest/smoothest boundary is given by Scherk's surface with $\beta = 0.4$, resulting in the fastest convergence. Both the sine-boundaries as well as our piece-wise initial condition do have a few non-smooth points (the sine on the edges, the piece-wise on the `bottom` and `top`-boundary), and thus have comparable convergence. Given the fact, that the order of convergence is quite similar for all testcases, we can conclude, that the number of iterations depends significantly on the initial residual. Typically, the residual decreases monotonically, though for some testcases, there is a short increase in the beginning. This might be improved by decreasing the relaxation parameter in the input file.

Furthermore, tests revealed, that the Laplace initial guess greatly improved convergence with respect to the other two options (since it decreases the initial residual by pre-shaping the curvature), and in some cases, is required to converge at all. We do not present the results here.



(a) Jacobian by AD by hand (equivalent to symbolic)



(b) Matrix-free Jacobian

Figure 12: Convergence results for the residual $\|r_k\|_2 = \|F^h\left[z_k^h\right]\|_2$ depending on different choices of the right-hand-side; all using the solution of the Laplace equation as a initial guess

4.5. Timings

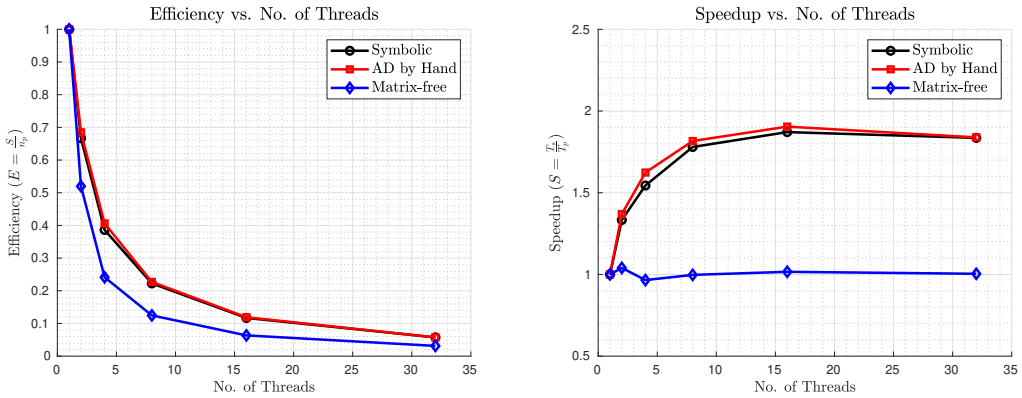


Figure 13: Parallel Efficiency and Speedup plots for different Jacobian methods

The code was timed to check the efficiency of the OpenMP parallelization. It can be seen, that for N = 1000, both Symbolic and AD by Hand take up as much time as the 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 Newton iteration of the matrix-free algorithm calls `dco` a large number of times (# iterations for the linear solve), and each `dco`-call is costly. On the other hand, this has the effect, that less Newton iterations are needed in general, since every Newton iteration effectively uses a new linearization of the Jacobian for each iteration of the linear solver.

As an example set, for N=500, tolerance for the residual inside the Newton iterations 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, while we use Newton's tolerance as 10^{-3} and linear solver tolerance as 10^{-7}, the execution time is 155.364 [s]. This is approximately half of the time compared to the previous case. For both these cases, we get the same error measured in both ℓ_2-norm and max-norm.

5. Conclusion and outlook

This last section should summarize the conclusions that we were able to draw from working on this project, and provides a short outlook on what could be still done to improve the software.

The main take-away messages, we were able to draw from the project, are:

- We were given the chance, to build a numerical software from scratch. This essentially forced us, to mobilize knowledge from most of the (mandatory) courses so far the Simulation Sciences curriculum. This way we saw, that we are already able to apply the theory from our studies (in some extent) to a toy problem (that could easily be extended to a practical one).
- The complexity of the project, though moderate, but fitted in a quite narrow time-frame required a good project management amongst the group, in case one wanted to implement everything, that was required. We learnt to appreciate platforms for digital collaboration, such as `git`, to realize working in a team on a software project.

Although our code fulfils the minimum requirements, there are a few points, that could be improved, and give rise to an outlook, what could be done in a second iteration:

- The 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
- Properly investigate the convergence behaviour of the method, and connect it to the smoothness of the boundary.
- Development of a more flexible grid, that allows (i) piece-wise boundary condition, that rely not on a parser, but on flexibly choosable boundary sections. I might furthermore be extended to arbitrary geometries, which requires a proper meshing algorithm, and different shapes of the elements. This would be simplified substantially then if one would use Finite Elements for the discretization.

References

[1] Wikipedia contributors. Minimal surface — Wikipedia, the free encyclopedia, 2019. [Online; accessed 28-February-2020].

[2] Zongsong Gan, Mark D. Turner, and Min Gu. Biomimetic gyroid nanostructures exceeding their natural origins. *Science Advances*, 2(5), May 2016.

[3] Frei Otto | German architect.

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

[5] Wikipedia contributors. Scherk surface — Wikipedia, the free encyclopedia, 2018. [Online; accessed 28-February-2020].

[6] 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 made to run on the RWTH cluster.[13]

Update as of last week of February: During the last week of February, the following options became necessary for the code to compile. As of February 28, it works again without, but if not, load the following options:

```
module switch intel gcc
module load LIBRARIES
module load intelmkl
```

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[14]. If one is interesting in the convergence, one can find the residual data over iteration inside `residual/`, which can be also plotted using Paraview.

[13] using either the current versions of the Intel (19) or GCC (9) compiler.
[14] to use this on the cluster, execute `module load GRAPHICS paraview` beforehand.