Vienna University of Technology

Computational Mathematics - Seminary

Institute of Analysis and Scientific Computing

PDE Constrained Shape Optimization

Authors:
Camilo Tello Fachin
12127084

Paul Genest
12131124

Supervisor:
Dr. Kevin Sturm

October 13, 2022

TECHNISCHE
UNIVERSITÄT
WIEN
Vienna University of Technology

Abstract

In the underlying work, a complete shape optimization procedure is done with The python Finite Element Method library NGSolve. As an initial problem, the linear stationary Stokes flow around a cylinder in a rectangular domain is considered. Since the velocity vectorfield is available as a solution of the weak stokes problem, the energy dissipation $J(\Omega)$ can be evaluated over the entire domain as an integral. The goal of the optimization problem is to minimize said energy dissipation, by perturbing the domain. The formulation of the functional $J(\Omega)$ and proof of existence of its derivative with respect to the domain perturbation $\mathrm{d}J(\Omega)(X)$ is sufficient, to formulate an augmented Lagrangian. The derivative of the Augmented Lagrangian can be used to minimize the Augmented Lagrangian, which is an approximative solution to the initial minimization of $J(\Omega)$. Additionally, since the minimization is done iteratively, one solves another PDE problem on the same domain which is the auxiliary problem. Its result is the perturbation of the domain in minimizing direction of $J(\Omega)$. The auxiliary problem is posed in such a way, that each perturbation is near conformal, or of near angle preserving character. This near conformality results in a mesh that is still of acceptable quality after hunderds of iterations. The succesful implementation in NGSolve with acceptable convergence behaviour and good agreement with literature are documented in this seminary paper. The agreement with the literature being, that the shape of the cylinder is actually reshaped to an ogive with 45 degree tips and the near conformality of the mesh over all iterations is guaranteed.

Contents

1 Introduction

This document was created in the context of the Computational Mathematics Seminary at the Technical University of Vienna (SE - 3ECTS). In this introduction, the scientific relevance of the work is highlighted and a brief overview of the topics covered is given.

PDE Constrained Shape Optimization is a topic of interest in almost all engineering fields where the relevant phenomena can be described by Partial Differential Equations (PDEs) and an optimization problem can be formulated. Here, the stationary linear stokes equations are the PDEs and the energy dissipation over the domain is optimized. The optimization can be described by a minimization problem which can be solved with the gradient descent method. Its important to note, that the PDE constraints and optimization goals used here, can be exchanged with arbitrary PDE constraints and optimization goals. Some parts, especially the proof for shape derrivative existence, can get more complex for e.g. Non-Linear PDEs or Transient PDEs.

NGSolve

The practical implementation part is done in NGSolve, which is a object oriented python Finite Element Method library with automatic differentiation capabilites. The mathematical formulations in this document can mostly be implemented directly, as one can see in either in appendix APPENDIX or in the associated Jupyter Notebook file. Gangl et. al. [3] have shown the PDE Constrained Optimization capabilites of NGSolve. For more in-depth explanations and examples on NGSolve visit ngsolve.org.

Minimization Problem

A generic PDE constrained optimization problem is of the following form:

$$\begin{aligned} &\min_{\Omega \in \mathcal{A}} J(\Omega, u) \\ s.t. \quad &\mathrm{B}_\Omega(u) = 0 \end{aligned}$$

Where Ω is the Domain for the PDE, $\mathcal{A}$ is the set of admissible shapes $J(\Omega, u)$ is a functional that is to be minimized and $\mathrm{B}_\Omega(u)$ is the PDE constraint and its solution u. The Domain Ω is what is going to be optimized in the underlying work.

Shape Derrivative

In order to find a numerical solution to the minimization problem with the gradient descent method, the existence of the analytical shape derrivative needs to be shown. Here the differentiability of $J(\Omega, u)$ at $\Omega \in \mathcal{A}$ in direction X is shown. As Sturm et. al. [2] have shown, the functional $J(\Omega, u)$ can be reduced to a functional $J(\Omega)$ and the shape derrivative $\mathrm{d}J(\Omega)(X)$ exists. In chapter **??**, the proof is recapitulated briefly.

Auxiliary Problem - Descent Direction

To find the gradient descent direction, here the vectorfield $-X$, an auxiliary problem needs to be solved. Since its solved with the Finite Element Method libary NGSolve, the PDE problem is posed in a weak sense where H is e.g. a Sobolov space. Find $X \in [H(\Omega)]^2$:

$$\mathrm{d}J(\Omega)(X) = \mathrm{b}(X, \varphi)_H \quad \forall \varphi \in H$$

If the bi-linear form $\mathrm{b}(.,.)_H$ is chosen such that it is positive definite, the negative solution X of the auxiliary problem points in the negative direction of the gradient.

Optimization Steps

The problem $\mathrm{B}_\Omega(u) = 0$ can be solved, the shape derivative $\mathrm{d}J(\Omega)(X)$ can be calculated, the auxilary problem which yields the descent direction $-X$ can be solved as well. In the last step the optimization takes place where one can e.g. use a gradient descent method from `numpy`.

$$X \in [C^{0,1}(\Omega)]^2 \quad , \mathrm{T}_t(.) := \mathrm{id} + tX \quad , \text{choose } t \in \mathbb{R}$$

Since we chose the gradient descent direction to point in $-X$ direction, the following must hold true for the energy dissipation functional:

$$J(\mathrm{T}_t(\Omega)) < J(\Omega)$$

Penalty Method

If the differentiability of the shape functional has been shown, a method to obtain a numerical result to the minimization problem, is by introducing an Augmented Lagrangian, here briefly discussed is the underlying work used Quadratic Penalty Method. Again one considers:

$$\min_{\Omega} J(\Omega) \quad \text{s.t.} \quad \mathrm{B}_\Omega(u) = 0$$

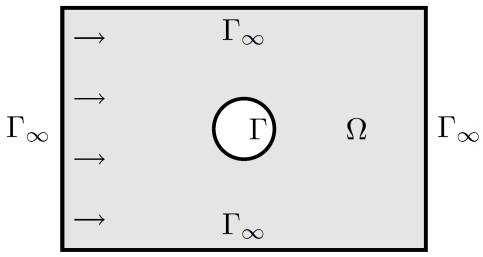
where $J(.) : \mathbb{R}^n \to \mathbb{R}$, and $\mathrm{B}_\Omega(.) : \mathbb{R}^n \to \mathbb{R}^p$ are differentiable functions. If the differentiability has been shown, the quadratic penalty method yields an approximative minimization result:

$$\mathcal{L}_\alpha(\Omega) = J(\Omega) + \frac{\alpha}{2}|\mathrm{B}_\Omega(u)|^2 \,, \quad \alpha > 0s$$

For more in depth elaborations on Penalty Methods and Augmented Lagrangian, see Numerical Optimization Lecture Notes from Dr. K. Sturm [5]. This Augmented Lagrangian can now be derived and used for the step direction $-X$.

2 The Stokes Equations in NGSolve

The Stokes Equations are linear partial differential equations, which describe a stationary incompressible Newtonian fluid flow with high viscosities and low Reynolds numbers. For the implementation in NGSolve, a suitable geometry and boundary conditions are the ones proposed by Sturm et. al. [2], where the fluid flow around a cylinder is investigated while the outer boundary of Ω is prescriped a velocity striclty in x direction, the so called far field velocity:



$$\begin{aligned} -\mu\Delta u + \nabla p &= 0 && \text{in } \Omega, \\ \operatorname{div} u &= 0 && \text{in } \Omega, \\ \mathbf{u} &= 0 && \text{on } \Gamma, \\ \mathbf{u} &= \mathbf{u}_\infty && \text{on } \Gamma_\infty, \end{aligned} \tag{1}$$

Figure 1: Domain Ω for Stokes PDE's (1) [2]

Where $\mu \in \mathbb{R}$ is the viscosity constant and is set to 1 for simplicity. The problem yields the vectorial velocity field $u : \Omega \to \mathbb{R}^d$ and the scalar pressure field $p : \Omega \to \mathbb{R}$. In order to solve the Stokes equation with the Finite Element Method in NGSolve, it needs to be transformed to the weak formulation, where the solutions u and p are linear combinations of basis functions in a Sobolev space. See Faustmann[4] Chapter 3 for further elaborations on Sobolev spaces. The weak formulation can be derrived by multipling the now called trial-functions u and p with test-functions v and q, perform transformations and integrate them. The test-functions have to fulfil certain conditions to permit the transformations in order to arrive at a weak problem with linear convergence rates, see Faustmann [4]:

Find $u \in [H_0^1(\Omega)]^d$ and $p \in L^2(\Omega)$ such that

$$\begin{aligned} \int_\Omega \nabla \mathbf{u} : \nabla \mathbf{v}\, dx + \int_\Omega \operatorname{div}(\mathbf{v}) p\, dx &= 0 \quad \forall \mathbf{v} \in [H_0^1(\Omega)]^d \\ \int_\Omega \operatorname{div}(\mathbf{u}) q\, dx &= 0 \quad \forall q \in L^2(\Omega) \end{aligned} \tag{2}$$

Instead of considering this as a system of equations, one can look at the mixed method as one variational problem on the product space $[H_0^1(\Omega)]^d \times L^2(\Omega)$, this is done by just adding both problems [4]:

$$\int_\Omega \nabla \mathbf{u} : \nabla \mathbf{v}\, dx + \int_\Omega \operatorname{div}(\mathbf{v}) p\, dx + \int_\Omega \operatorname{div}(\mathbf{u}) q\, dx = 0 \quad \forall (\mathbf{v}, q) \in [H_0^1(\Omega)]^d \times L^2(\Omega) \tag{3}$$

In lines 19-26 of listing 2, the variational problem 3 is added to a `BilinearForm()`. After assembling of the system, in line 27-31 the non-zero Dirichlet conditions are assigned. When setting up the geometry, the boundaries already have to be named to do the boundary conditions assignment. The geometry shown in figure 1, is defined in the beginning in lines 5-11.

Basic Stokes PDE's with NGSolve in Python

```
k = 2
V = H1(mesh,order=k, dirichlet="top|bot|cyl|in|out")
Q = H1(mesh,order=k-1)
FES = FESpace([V,V,Q])
ux,uy,p = FES.TrialFunction()
vx,vy,q = FES.TestFunction()
def Equation(ux,uy,p,vx,vy,q):
    div_u = grad(ux)[0]+grad(uy)[1] # custom divergence u
    div_v = grad(vx)[0]+grad(vy)[1] # custom divergence v
    return (grad(ux)*grad(vx)+grad(uy)*grad(vy) + div_u*q + div_v*p)* dx
a = BilinearForm(FES)
a += Equation(ux,uy,p,vx,vy,q)
a.Assemble()
gfu = GridFunction(FES)
uinf = 0.001
uinf_c = CoefficientFunction((uinf))
gfu.components[0].Set(uinf_c, definedon=mesh.Boundaries("in|top|bot|out"))
def solveStokes():
res = gfu.vec.CreateVector()
res.data = -a.mat * gfu.vec
inv = a.mat.Inverse(FES.FreeDofs())
gfu.vec.data += inv * res
scene_state.Redraw()
```

Below the solution obtained with NGSolve (see listing 2). On the surface of the cylinder, the no-slip condition (standard Dirichlet = 0) can be observed. Also an intuitive observation of the fulfilled continuity can be made: where the cross section is smaller, e.g. in x vicinity of the cylinder, the velocity is increased:

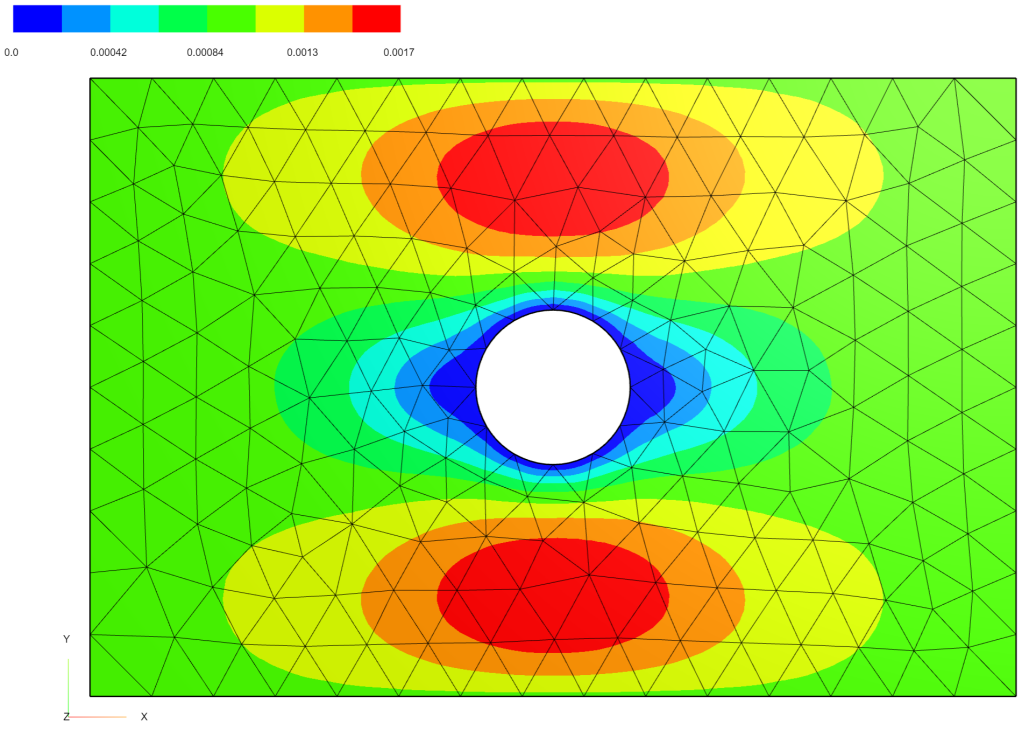


Figure 2: Surface Plot - Velocity $||\mathbf{u}||_2$ of Stokes Flow - FEM solution to problem (1)

3 Shape Derivative - Differentiability

As explained in the introduction, the existence of the shape derivative needs to be shown. The perturbations of the shape Ω are described by the following transformation: $\Omega_t := (Id + tX)(\Omega)$ where. For small perturbations and $t > 0$, the shape derivative is: [3]

$$DJ(\Omega)(X) := \left(\frac{\partial}{\partial t} J(\Omega_t)\right)\Big|_{t=0} = \lim_{t\to 0} \frac{J(\Omega_t) - J(\Omega)}{t} \tag{4}$$

This notion of the shape derivative is used in this chapter in the context of differentiability. The functional that returns a scalar quantity represetative of the energy dissipation is shown here where : is the Frobenius product and D$\mathbf{u}$ is the Jacobi matrix of $\mathbf{u}$.

$$J(\Omega) = \frac{1}{2}\int_\Omega \mathrm{D}\mathbf{u} : \mathrm{D}\mathbf{u}\,\mathrm{dx} \tag{5}$$

Sturm et. al. [2] proposed the shape derrivative of following form is given by:
Shape Derivative of J at Ω in direction $X \in [C^{0,1}(\bar{\Omega})]^2$

$$\mathrm{d}J(\Omega)(X) = \int_\Omega \mathrm{S}_1 : \mathrm{D}X\,\mathrm{dx} \tag{6}$$

$$\mathrm{S}_1 = \left(\frac{1}{2}\mathrm{D}\mathbf{u} : \mathrm{D}\mathbf{u} - p\,\mathrm{div}(\mathbf{u})\right)\mathrm{I}_2 + \mathrm{D}\mathbf{u}^\top p - \mathrm{D}\mathbf{u}^\top \mathrm{D}\mathbf{u} \tag{7}$$

where $(\mathbf{u}, p)$ solve (3)

Proof. *let $X \in [C^{0,1}(\bar{\Omega})]^2$ with $X = 0$ on Γ_∞ be a given vectorfield.*
Set $\mathrm{T}_t(.) := \mathrm{id} + tX$, with $t \in \mathbb{R}$ and $\Omega_t := \mathrm{T}_t(\Omega)$, where $(\mathbf{u}_t, p_t)$ solve (3) and Ω is replaced by Ω_t s.t. $p_t \in \mathrm{L}^2(\Omega_t), \int_{\Omega_t} p_t\,\mathrm{dx} = 0$ and $\mathrm{u}_t \in [H^1(\Omega_t)]^2$:

$$\int_{\Omega_t} \mathrm{D}\mathbf{u}_t : \mathrm{D}\mathbf{v} + \mathrm{div}(\mathbf{v})\,p_t + \mathrm{div}(\mathbf{u}_t)\,q\,dx = 0 \quad \forall (v, q) \in [H_0^1(\Omega_t)]^d \times L^2(\Omega_t) \tag{8}$$

Introduction of change of variables shows that $(\mathbf{u}^t, p^t) := (\mathbf{u}_t \circ \mathrm{T}_t, p_t \circ \mathrm{T}_t)$ satisfy:

$$\begin{gathered}\int_\Omega \det(\mathrm{DT}_t)\left(\mathrm{DT}_t^{-1}\mathrm{D}\mathbf{u}^t : \mathrm{DT}_t^{-1}\mathrm{D}\mathbf{v} - p\,\mathrm{tr}(\mathrm{D}\mathbf{v}\mathrm{DT}_t^{-1}) + q\,\mathrm{tr}(\mathrm{D}\mathbf{u}\mathrm{DT}_t^{-1})\right)\mathrm{dx} \\ \forall (v, q) \in [H^1(\Omega)]^2 \times L^2(\Omega)\end{gathered} \tag{9}$$

with:

$$\begin{aligned}\mathrm{D}\mathbf{v} \circ \mathrm{T}_t &= \mathrm{D}(\mathbf{v} \circ \mathrm{T}_t) \\ \mathrm{div}(\mathbf{v}) &= \mathrm{tr}\left(\mathrm{D}(\mathbf{v} \circ \mathrm{T}_t)(\mathrm{DT}_t^{-1})\right)\end{aligned}$$

The functional $J(\Omega, \mathbf{u})$ is now reduced to the functional $J(\Omega)$, since the change of the quantities $(\mathbf{u}, p)$ is taken into account by the transformation theorem. The minimum of (5) satisfies the saddlepoint problem (9). It can be obtained with the Lagrange Multiplier method, see Faustmann [4]. The corresponding Lagrangian which can be used to minimize (5) is:

$$\begin{aligned}\mathcal{L}(t, \mathbf{v}, q) = \frac{1}{2} \int_\Omega \det(\mathrm{DT}_t) \mathrm{D}\mathbf{v} (\mathrm{DT}_t)^{-1} : \mathrm{D}\mathbf{v} (\mathrm{DT}_t)^{-1} \,\mathrm{dx} \\ - \int_\Omega \det(\mathrm{DT}_t) q \,\mathrm{tr}\left(\mathrm{D}\mathbf{v} (\mathrm{DT}_t)^{-1}\right)\end{aligned} \tag{10}$$

To find the shape derivative, one can now derive this parametrized Lagrangian, for details on the derivation of parametrized Lagrangians, see K. Ito et. al. [1]. With the derivative of the Lagrangian obtained, it holds true that:

$$\mathrm{d}J(\Omega)(X) = \frac{\mathrm{d}}{\mathrm{d}t} \mathcal{L}(t, \mathbf{u}^t, 0)\big|_{t=0} = \frac{\partial}{\partial t} \mathcal{L}(0, \mathbf{u}, p) = \int_\Omega \mathrm{S}_1 : \mathrm{D}X \,\mathrm{dx} \tag{11}$$

□

4 Augmented Lagrangian and Geometric Constraints

In the previous chapter, the differentiability of the shape function with respect to the perturbation parameter t was recapitulated [2]. One can now employ a minimization problem which yields an approximative solution. This is where the Augmented Lagrangian is introduced. Since differentiability was shown, one can show that the negative derrivative of the Augmented Lagrangian is an approximate minimizer to the Lagrangian (10), for elaborations on Augmented Lagrangians see Sturm Lecture Notes [5]. One considers again the Lagrangian (10) in the unparametrized way which can be evaluated on the FEM mesh:

$$\mathcal{L}(\Omega) = \frac{1}{2}\int_\Omega \mathrm{D}\mathbf{u} : \mathrm{D}\mathbf{u}\,\mathrm{dx} + \int_\Omega \nabla\mathbf{u} : \nabla\mathbf{u}\,\mathrm{dx} + \int_\Omega \mathrm{div}(\mathbf{u})p\,\mathrm{dx} + \int_\Omega \mathrm{div}(\mathbf{u})p\,\mathrm{dx} \tag{12}$$

If one minimizes this problem, one can observe: the numerical scheme will just make the obstacle smaller since this will result in a minimization of the drag as well. This is not a trivial result but also not a result of interest. Therefore one analyzes the volume and barycenter of the obstacle and formulates terms that penalize deviations from the initial values of the volume and barycenter.

$$\mathrm{vol}(\Omega) = \int_\Omega 1\ \mathrm{dx} \in \mathbb{R} \tag{13}$$

$$\mathcal{L}^2_{\mathrm{vol}}(\Omega) = \alpha\ \Big(\mathrm{vol}(\mathbf{T}_t(\Omega)) - \mathrm{vol}(\Omega_0)\Big)^2 \tag{14}$$

The quantity $\mathcal{L}_{\mathrm{vol}}(\Omega)$ is a quadratically penalized term that is easy to derrive and is used to formulate the Augmented Lagrangian. If the deformed domain Ω yields an obstacle of identical volume, the term is zero, deviations from the initial obstacle volume are penalized. The same is done for the barycenter of the obstacle.

$$\mathrm{bc}(\Omega) = \frac{1}{\mathrm{vol}(\Omega)}\int_\Omega \mathbf{x}\ \mathrm{d}\mathbf{x} \in \mathbb{R}^2 \tag{15}$$

Since it is a vectorial quantity, the integral is decomposed into its components such that it yields a scalar quantity:

$$\mathrm{bc}_x(\Omega) = \frac{1}{\mathrm{vol}(\Omega)}\int_\Omega x\ \mathrm{dx}\,, \quad \mathrm{bc}_y(\Omega) = \frac{1}{\mathrm{vol}(\Omega)}\int_\Omega y\ \mathrm{dx} \tag{16}$$

$$\mathcal{L}^2_{\mathrm{bc}}(\Omega) = \beta\Big(\mathrm{bc}_x(\mathbf{T}_t(\Omega)) - \mathrm{bc}_x(\Omega_0)\Big)^2 + \gamma\Big(\mathrm{bc}_y(\mathbf{T}_t(\Omega)) - \mathrm{bc}_y(\Omega_0)\Big)^2 \tag{17}$$

Finally, the quadratically penalized Augmented Lagrangian:

$$\mathcal{L}_{\mathrm{aug}}(\Omega) = \mathcal{L}(\Omega) + \mathcal{L}^2_{\mathrm{vol}}(\Omega) + \mathcal{L}^2_{\mathrm{bc}}(\Omega) \tag{18}$$

The parameters α, β and γ are used to weigh the quadratically penalized terms and vary them dynamically while iterating.

Remark: The quadratically penalized terms do not evaluate the surface and barycenter of the obstacle and its deviations, but rather of the entire domain Ω, but since the obstacle is in the center and the entire geometry symmetric, the obtained quantities are representative of the surface and barycenter of the obstacle as well.

5 Shape Derivative in NGSolve

5.1 Derrivative of the Augmented Lagrangian

After the Augmented Lagrangian is set up accordingly, the derivatives $\mathrm{d}\mathcal{L}_{\mathrm{i}}(\Omega)(X)$ can be formulated and implemented. In the NGSolve implementation, analytical derivatives for the terms $\mathrm{d}\mathcal{L}^2_{\mathrm{bc}}(\Omega)(X)$ and $\mathrm{d}\mathcal{L}^2_{\mathrm{vol}}(\Omega)(X)$ are directly used. Here the definition for $\mathrm{vol}(\Omega)$ is equal to the definition in eq. (13) and $\mathrm{bc}_{x,y}(\Omega)$ are equal to the definitions in eq. (16).

Derivative for the volume constraint term in X direction:

$$\mathrm{d}\mathcal{L}^2_{\mathrm{vol}}(\Omega)(X) = 2\,\alpha\Big(\big(\mathrm{vol}(\Omega) - \mathrm{vol}(\Omega_0)\big)\Big)\,\mathrm{div}(X) \tag{19}$$

Derivative for the barycenter constraint term in X direction:

$$\begin{aligned}\mathrm{d}\mathcal{L}^2_{\mathrm{bc}}(\Omega)(X) = {} & 2\,\beta\Big(\mathrm{bc}_x(\Omega) - \mathrm{bc}_x(\Omega_0)\Big)\int_\Omega \frac{1}{\mathrm{vol}(\Omega)^2}\,\mathrm{div}(X)\,x + \frac{1}{\mathrm{vol}(\Omega)}\,\mathrm{div}(X)\,x\cdot\vec{e}_x\,X\,\mathrm{dx} \\ & + 2\,\gamma\Big(\mathrm{bc}_y(\Omega) - \mathrm{bc}_y(\Omega_0)\Big)\int_\Omega \frac{1}{\mathrm{vol}(\Omega)^2}\,\mathrm{div}(X)\,y + \frac{1}{\mathrm{vol}(\Omega)}\,\mathrm{div}(X)\,y\cdot\vec{e}_y\,X\,\mathrm{dx}\end{aligned} \tag{20}$$

To obtain the term $\mathrm{d}\mathcal{L}(\Omega)(X)$, the NGSolve command `DiffShape(X)` is used, to utilize the library's Automatic Differentiation capabilites. Finally one arrives at the shape derivative that is, to reiterate, the derivative of the Augmented Lagrangian:

$$\mathrm{d}\mathcal{L}_{\mathrm{aug}}(\Omega)(X) = \mathrm{d}\mathcal{L}(\Omega)(X) + \mathrm{d}\mathcal{L}^2_{\mathrm{vol}}(\Omega)(X) + \mathrm{d}\mathcal{L}^2_{\mathrm{bc}}(\Omega)(X) \tag{21}$$

In the listing below, the implementation in NGSolve, for more details see either appendix APPENDIX REFERENCE or the `JupyterNotebook` :

Derivative of Augmented Lagrangian

```
# Initilize LinearForm object
dLOmega = LinearForm(VEC)
# add automatic shape differentiation term to LinearForm object
dJOmega += Lagrangian.DiffShape(X)
# add analytically derrived volume constraint term to LinearForm object
vol = Parameter(1)
vol.Set( Integrate ( surf_t ,mesh))
alpha0 = 1e-4
alpha = Parameter(alpha0)
dLOmega += 2*alpha*(vol-surf_0)*div(X)*dx
# add analytically derrived x-barycenter constraint term to LinearForm object
beta0 = 1e-3
beta = Parameter(beta0)
bc_x = Parameter(1)
bc_x.Set((1/surf_0)*Integrate(bc_tx,mesh))
dJOmega += 2*beta*(bc_x-bc_0x)*((1/vol**2)*div(X)*x+(1/vol)*div(X)*x*sum(gfset.vecs[0].data)[0])*dx
# add analytically derrived y-barycenter constraint term to LinearForm object
bc_y = Parameter(1)
bc_y.Set((1/surf_0)*Integrate(bc_ty,mesh))
dJOmega += 2*beta*(bc_y-bc_0y)*((1/vol**2)*div(X)*y+(1/vol)*div(X)*y*sum(gfset.vecs[0].data)[1])*dx
```

5.2 Auxiliary Problem

The descent direction in the gradient descent method, is the direction $-X$. In the previous chapter, a formulation for $\mathrm{d}\mathcal{L}_{\mathrm{aug}}(\Omega)(X)$ was derived. As a next step one needs to formulate an auxiliary problem: since it is demanded, that the functional $\mathcal{L}_{\mathrm{aug}}(\Omega)$ is minimized, one follows the negative gradient $-X$. This yields a PDE which can be solved in a weak sense with FEM in NGSolve. The weak formulation of the auxiliary problem for X reads as follows:

find $X \in [H(\Omega)]^2$ such that:

$$\mathrm{d}\mathcal{L}_{\mathrm{aug}}(\Omega)(X) = -(X, \varphi)_H \quad \forall \varphi \in H(\Omega) \tag{22}$$

Sturm et. al. [2] have proposed different spaces H for the auxiliary problem and investigated their impact analytically. The main criterion being, that the bi-linear form $(X, \varphi)_H$ is positive definite, to guarantee that the direction $-X$ is indeed negative. For the case $H = H^1(\Omega)$:

$$(X, \varphi)_{H^1(\Omega)} = \int_\Omega X\,\varphi + \mathrm{D}X : \mathrm{D}\varphi \,\mathrm{dx} \tag{23}$$

5.3 The Vectorfield X as a Conformal Mapping

For the notion of conformality, one introduces the Cauchy-Riemann Equations. This is used here, to omit remeshing of the domain or severe degradation of the elements. This because remeshing is relatively expensive computation wise and element degradation would lead to a large local error of the Stokes flow solution. If the vectorfield X satisfies the Cauchy-Riemann Equations it is a holomorphic injective transformation, which is conformal. Where $X = (X_1, X_2) \in [C^1(\Omega)]^2$:

$$\begin{aligned} \partial_x X_1 &= \partial_y X_2 \\ \partial_y X_1 &= -\partial_x X_2 \end{aligned} \tag{24}$$

The linear operator $\mathcal{B}$ is used for more compact notation of the Cauchy-Riemann equations (24):

$$\mathcal{B} = \begin{pmatrix} -\partial_x & \partial_y \\ \partial_y & \partial_x \end{pmatrix}, \quad [C^1(\Omega)]^2 \to [C^0(\Omega)]^2 \tag{25}$$

With introduction of $\mathcal{B}$, one can write the CR-Equations (24) as:

$$\mathcal{B}X = 0$$

Now one can use the CR-Equations to adjust the auxiliary problem to yield nearly conformal mappings. Where $||\,.\,||_P : P \to \mathbb{R}$ is a norm on a Hilbert space P and $\mathcal{B} : H \to P$ is a linear continuous operator in P such that $\mathcal{B}(H) \subset P$, find $X \in [H(\Omega)]^2$

$$\mathrm{d}\mathcal{L}_{\mathrm{aug}}(\Omega)(X) = \alpha(\mathcal{B}X, \mathcal{B}\varphi)_P + (X, \varphi)_H, \quad \alpha \in \mathbb{R}, \quad \forall \varphi \in [H(\Omega)]^2 \tag{26}$$

The parameter α can now be used to weigh the conformality of the mapping. For higher α the mapping is more conformal.

6 Iteration

Since the minimization is done in iterations, we have to keep track of the previous deformations. This is done in ngsolve by adding the parts of `gfX` to another variable called `gfset`. This `gfset` is then always used to call `SetDeformation()` on the mesh. With each call, this adds the `gfset` onto the mesh. To circumvent this, after each iteration `UnsetDeformation()` is called. To not overshoot anything, instead of adding the entire `gfX` to `gfset`, it is scaled by a number divided by its norm. That way we can make sure, that each iteration deforms the mesh in small, similar sized steps. There is still one inconsistency: the `gfX` can also deform nodes inside the mesh, which change nothing for the real solution, but count towards this norm. That problem is solvable, by integrating this `GridFunction` over its boundary. Since the outside of our square are dirichlet boundaries, this way only the changes on the obstacle are measured. Another important thing to be aware of is, that symmetric deformations around an obstacle might cancel each other out in the integral. This is circumvented by calculating with the squared values.
In this chapter we want do describe the process of one iterations which gets repeated time after time until the optimal shape is achieved.
There are a few parameters that are recalculated or conditionally updated through the iterations. This includes the current volume and barycenter values, their respective scalings α and β, and *gamma* for the Cauchy Riemann terms that ensure a better mesh quality.
Before we start with the iteration process we reset all possible variables: This includes the `gfset`, resetting the scene, and to reinitialize the parameters for the geometric constraints. If this step is skipped, one could start with weird variable values that lead to drastically different results.

Reset before iteration

```
1  gfset.Set((0,0))
2  mesh.SetDeformation(gfset)
3  scene.Redraw()
4  updateParams()
5  alpha0 = 1e-4
6  beta0 = 1e-0
7  gamma0 = 1e2
8  alpha.Set(alpha0)
9  beta.Set(beta0)
10 gamma.Set(gamma0)
11 iter_max = 750
```

This iteration is bounded by a maximum number of steps, even though it is also be possible to take some measure on the `gfX` and determine a stop by it. This still requires parameter tuning, especially because some measures could yield different results depending on the mesh width. Other possibilities for breaks in the loop are a maximum number for the scaling parameters e.g. α or a very small difference in the drag from one iteration to the next.
Each iteration then starts by calling `SetDeformation(gfset)`, and redrawing needed scenes. This would also be the place to gather data, like the current drag or area of the mesh, etc.
Afterwards we start calculating towards the next step: We Assemble the state equation bilinear form and solve it over this newly deformed mesh. Following this we assemble the linear and bilinear form for the shape derivative and solve for a new deformation. After this is done, one can already use `UnsetDeformation()` and the last thing to do is updating values.
We scale the `gfX` to its desired magnitude and subtract it from `gfset`. At this point it is also wise to check for some measure of close we are to a good solution and for example increase the parameters for the geometric constraints.

The entire code inside the loop looks like this:

Iteration

```
    mesh.SetDeformation(gfset)
    scene.Redraw()
    data.append(vol.Get())
    a.Assemble()
    solveStokes()
    b.Assemble()
    dLOmega.Assemble()
    SolveDeformationEquation()
    updateParams()
    mesh.UnsetDeformation()
    gfxnorm = Norm(gfX.vec)
    scale = 0.01 / gfxnorm

    if(gfxnorm < 1e-5):
      if alpha.Get() < 1:
        increaseParams(2,True)
      else:
        break
    gfset.vec.data -= scale * gfX.vec
```

It is also possible to implement some sort of line search for the step size, similar to the armijo rule. This way one can do less iterations but take better/bigger steps, being more efficient in regards to computational efforts. More to this can be read in ...

7 Results and Conclusion

7.1 General Results and Agreement with Literature

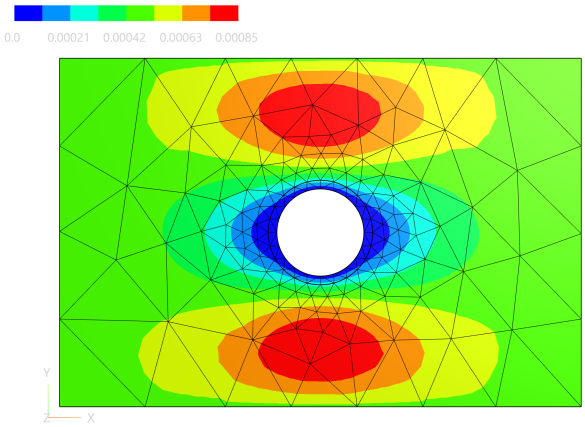


Figure 3: $||\mathbf{u}||_2$ on Ω_n for n = 0

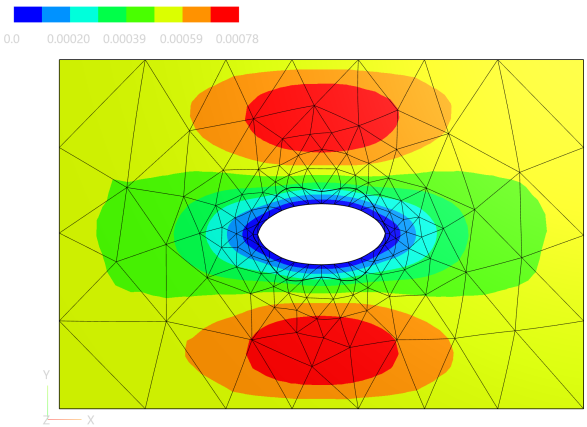


Figure 4: $||\mathbf{u}||_2$ on Ω_n for n = 800

In the figures 3 and 4 above, the initial and final domains Ω show good agreement with the results obtained by Sturm et. al. [3] where the tips of the ogive show approximately 45 degree angles. The constraint for near conformality posed in equation (26) also yields acceptable mesh quality at the tips, where the mesh withouth these contraints would result in a self intersecting mesh. This renders the Stokes solution $(\mathbf{u}, p)$ useless locally.

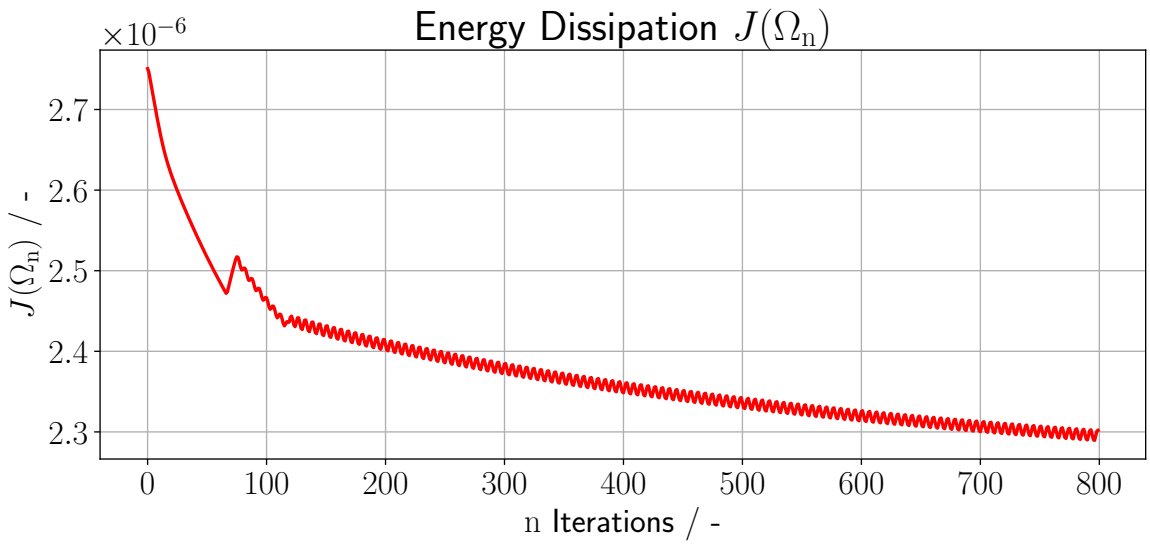


Figure 5: Evaluation of Energy Dissipation on Ω

In figure 5, convergence of the minimization problem can be observed. However, not only convergence with respect tho the energy minimization is relevant in the posed problem. Since the Augmented Lagrangian (18) also tracks the quantities $\text{vol}(\Omega_n)$, $\text{bc}_x(\Omega_n)$, $\text{bc}_y(\Omega_n)$ by keeping them constant or rather penalize deviations from the initial value, investigation of these values are also done subsequently.

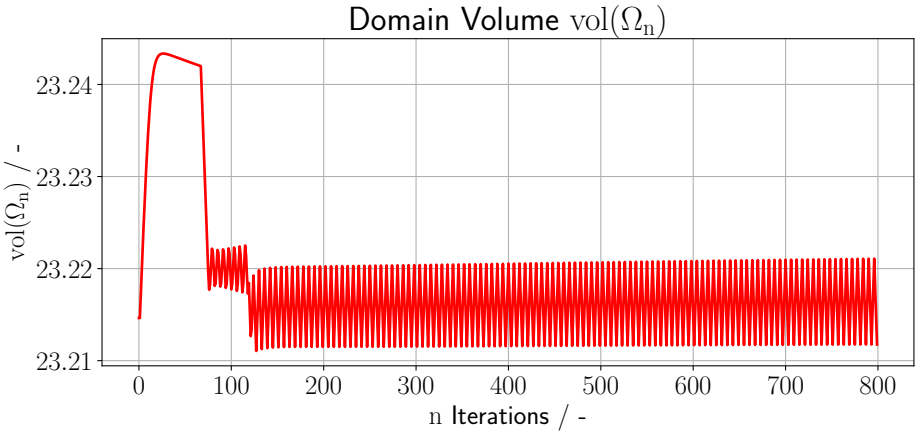


Figure 6: Volume of Ω

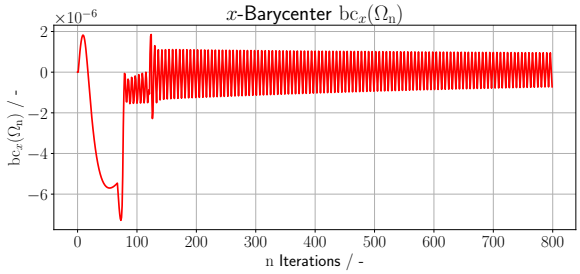


Figure 7: x Component of Barycenter of Ω

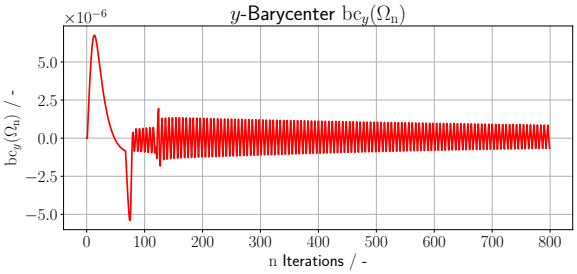


Figure 8: y Component of Barycenter of Ω

In figure 6, oscillations around the initial value $\mathrm{vol}(\Omega_0)$ can be be observed, which further underlines the approximative character of the Augmented Lagrangian Method (18). The same behaviour can be observed for the barycenters in x and y directions, which for a symmetric domain Ω with respects to the coordinate system should both be 0.

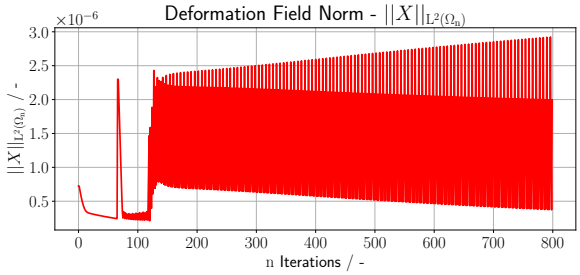


Figure 9: Norm of X on Domain

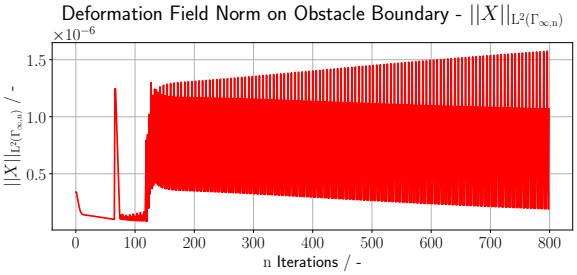


Figure 10: Norm of X on Boundary

In figures 9 and 10 the L^2 norms on the domain and the obstacle boundary are evaluated. These quantities have similar shape but a scaling factor of 2 in between. In a subsequent Seminary Project, one could investigate their scaling impact for the iterations further.

7.2 Cauchy-Riemann Constraint Impact on Mesh and Shape

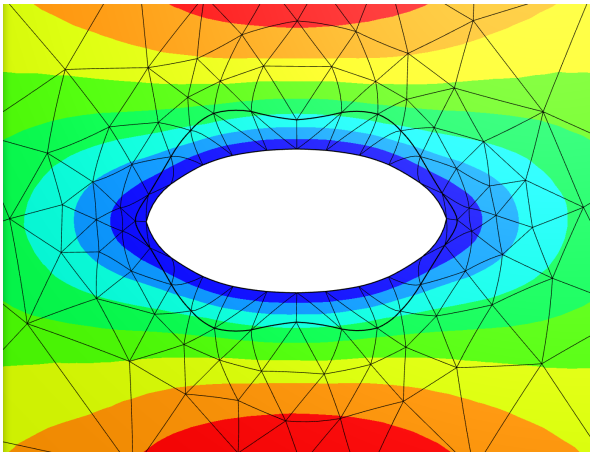


Figure 11: Final Ω with CR Constraint

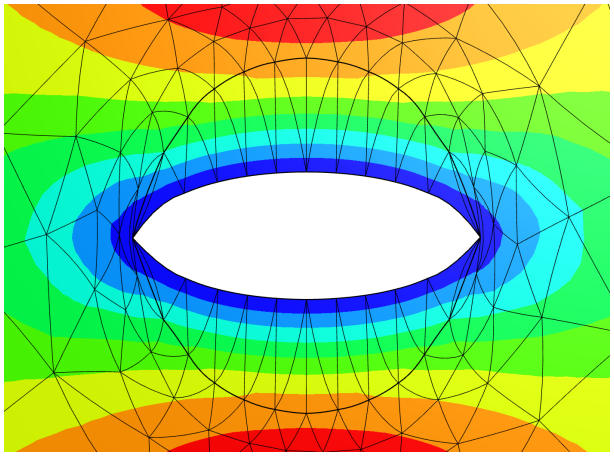


Figure 12: Final Ω without CR Constraint

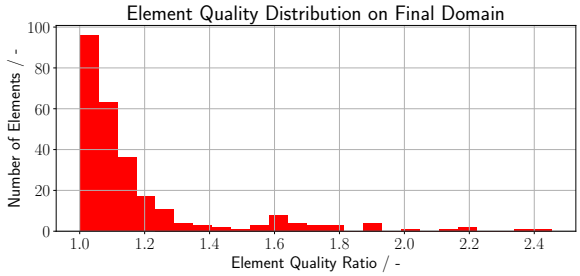


Figure 13: Element Quality with CR Constraint

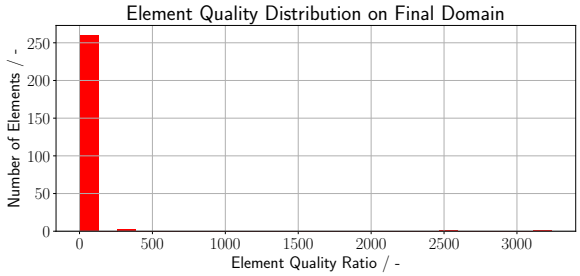


Figure 14: Element Quality without CR Constraint

In figures 11 - 14, the impact of the Cauchy-Riemann constraint can be observed. The mesh in figure 12 adjacent to the obstacle is highly distorted and the solutions $(\mathbf{u}, p)$ therefore not reliable anymore. The element quality at the tips is approximately 3000 without the CR constraint, as shown in figure 14. With applied CR constraint with $\alpha = 150$, the minimization after 800 iterations results in a acceptable element quality distribution, as shown in figures 11 and 13.

7.3 Cauchy-Riemann Constraint Impact on Convergence Behaviour

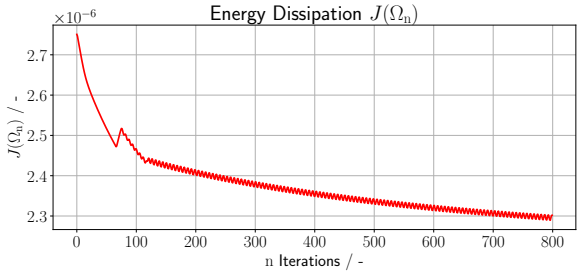


Figure 15: Energy Diss. with CR Constraint

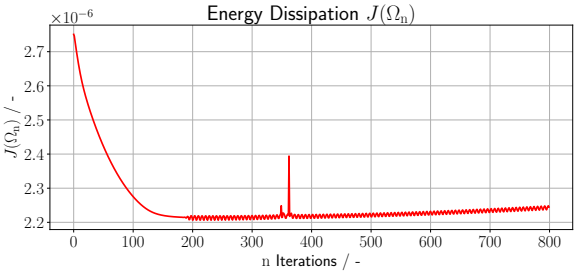


Figure 16: Energy Diss. without CR Constraint

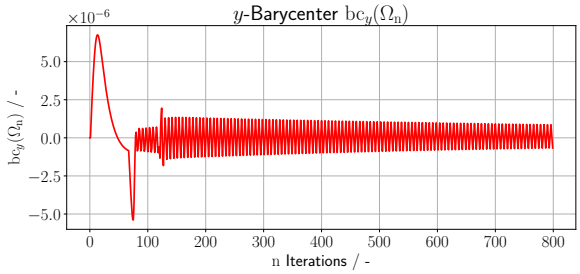


Figure 17: y-Barycenter with CR Constraint

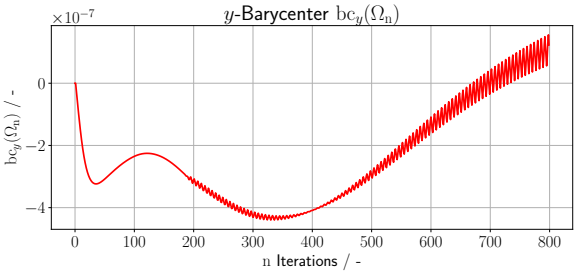


Figure 18: y-Barycenter without CR Constraint

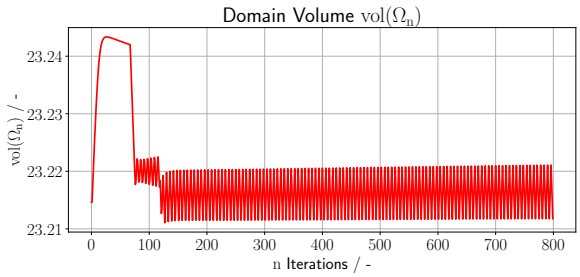


Figure 19: Volume with CR Constraint

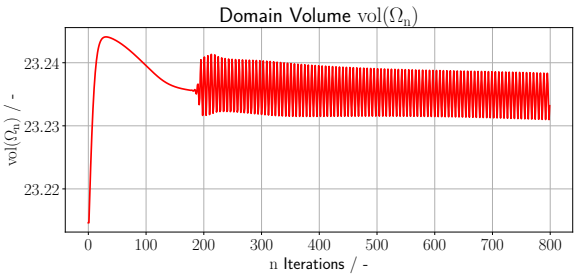


Figure 20: Volume wihtout CR Constraint

In figure 16, the energy dissipation increases again after 200 iterations, while the y-barycenter in figure 18 and the volume of Ω Figure 20 are not nearly in the vicinity of the desired initial values. For this PDE-Constrained Shape optimization setup, the CR-Equations are therefore even needed, since no convergence can be observed without them for all tracked quantities.

References

[1] Ito, Kazufumi, Kunisch, Karl, and Peichl, Gunther H., "Variational approach to shape derivatives," *ESAIM: COCV*, vol. 14, no. 3, pp. 517–539, 2008. DOI: 10.1051/cocv:2008002. [Online]. Available: https://doi.org/10.1051/cocv:2008002.

[2] J. Iglesias, K. Sturm, and F. Wechsung, "Two-Dimensional Shape Optimization with Nearly Conformal Transformations," *SIAM Journal on Scientific Computing*, vol. 40, A3807–A3830, Jan. 2018. DOI: https://doi.org/10.1137/17M1152711.

[3] P. Gangl, K. Sturm, M. Neunteufel, and J. Schöberl, "Fully and Semi-automated Shape Differentiation in NGSolve," *Structural and multidisciplinary optimization*, vol. 63, no. 3, pp. 1579–1607, 2021. DOI: https://doi.org/10.1007/s00158-020-02742-w.

[4] M. Faustmann and J. Schoeberl, "Lecture notes for Numerical Methods for PDE's - TU Vienna ASC," Jun. 2022.

[5] K. Sturm, "Lecture notes for PDE constrained Optimization - TU Vienna ASC," Jun. 2022.

A Python Code Listing

NGSolve Shape Optimization Code in Python

```
from ngsolve import *
from netgen.geom2d import SplineGeometry
from ngsolve.webgui import Draw
import numpy as np
import matplotlib.pyplot as plt

geo = SplineGeometry()
h_coarse = 1
h_fine = 0.15
geo.AddRectangle( (-3,-2), (3, 2), bcs = ("top", "out", "bot", "in"), leftdomain=1, rightdomain=0, maxh=
    h_coarse)
geo.AddCircle(c=(0, 0), r=0.6, leftdomain=2, rightdomain=1, bc="outer_cylinder", maxh=h_fine)
geo.AddCircle(c=(0, 0), r=0.5, leftdomain=0, rightdomain=2, bc="cyl", maxh=h_fine)
mesh = Mesh(geo.GenerateMesh(maxh=h_coarse))
mesh.Curve(2)

# Order of spaces for Taylor-Hood Elements
k = 2
# H1 vs VectorH1 -> vector field?!
V = H1(mesh,order=k, dirichlet="top|bot|cyl|in|out")
Q = H1(mesh,order=k-1)
FES = FESpace([V,V,Q]) # [V,Q] (without VectorH1)

ux,uy,p = FES.TrialFunction()
vx,vy,q = FES.TestFunction()

# stokes equation
def Equation(ux,uy,p,vx,vy,q):
    div_u = grad(ux)[0]+grad(uy)[1] # custom div
    div_v = grad(vx)[0]+grad(vy)[1]
    return (grad(ux)*grad(vx)+grad(uy)*grad(vy) + div_u*q + div_v*p)* dx

a = BilinearForm(FES)
a += Equation(ux,uy,p,vx,vy,q)
a.Assemble()

gfu = GridFunction(FES)
uinf = 0.0005
uin = CoefficientFunction((uinf))
gfu.components[0].Set(uin, definedon=mesh.Boundaries("in|top|bot|out"))
x_velocity = CoefficientFunction(gfu.components[0])
scene_state = Draw(x_velocity, mesh, "vel")

def solveStokes():
    res = gfu.vec.CreateVector()
    res.data = -a.mat * gfu.vec
    inv = a.mat.Inverse(FES.FreeDofs())
```

```
    gfu.vec.data += inv * res
    scene_state.Redraw()
solveStokes()

def calc_drag(gfu):
    ux = gfu.components[0]
    uy = gfu.components[1]
    return 0.5*(grad(ux)*grad(ux)+grad(uy)*grad(uy)).Compile()*dx

alpha = 1e-4
surf_t = CoefficientFunction(1)
surf_0 = Integrate(surf_t,mesh)

def calc_surf_change():
    return alpha*(surf_t*dx-surf_0)**2

bc_tx = CoefficientFunction(x)
bc_ty = CoefficientFunction(y)
bc_0x = 1/surf_0*Integrate(bc_tx,mesh)
bc_0y = 1/surf_0*Integrate(bc_ty,mesh)

# Test and trial functions for shape derivate
VEC = H1(mesh, order=2, dim=2, dirichlet="top|bot|in|out")
PHI, X = VEC.TnT()
# gfset denotes the deformation of the original domain and will be updated during the shape optimization
gfset = GridFunction(VEC)
gfset.Set((0,0))
mesh.SetDeformation(gfset)
SetVisualization (deformation=True)

ux = gfu.components[0]
uy = gfu.components[1]
p = gfu.components[2]

vol = Parameter(1)
vol.Set(Integrate(surf_t,mesh))
Lagrangian = Equation(ux,uy,p,ux,uy,p) + calc_drag(gfu)

dJOmega = LinearForm(VEC)
# automatic shape differentiation
dJOmega += Lagrangian.DiffShape(X)

# volume side constraint
vol = Parameter(1)
vol.Set(Integrate(surf_t,mesh))
alpha0 = 1e-4
alpha = Parameter(alpha0)
dJOmega += 2*alpha*(vol-surf_0)*div(X)*dx

# barycenter x sideconstraint
beta0 = 1e-3
```

```
98    beta = Parameter(beta0)
99    bc_x = Parameter(1)
100   bc_x.Set((1/surf_0)*Integrate(bc_tx,mesh))
101   dJOmega += 2*beta*(bc_x-bc_0x)*((1/vol**2)*div(X)*x+(1/vol)*div(X)*x*sum(gfset.vecs[0].data)[0])*dx
102
103   # barycenter y  sideconstraint
104   bc_y = Parameter(1)
105   bc_y.Set((1/surf_0)*Integrate(bc_ty,mesh))
106   dJOmega += 2*beta*(bc_y-bc_0y)*((1/vol**2)*div(X)*y+(1/vol)*div(X)*y*sum(gfset.vecs[0].data)[1])*dx
107
108
109   b = BilinearForm(VEC)
110   #b += (InnerProduct(grad(X),grad(PHI))).Compile() *dx + (InnerProduct(X,PHI)).Compile()*dx
111   b += (InnerProduct(grad(X),grad(PHI)+grad(PHI).trans)).Compile() *dx + (InnerProduct(X,PHI)).Compile()*dx
112
113   #Cauchy-Riemann Penalisation
114   gamma0 = 25
115   gamma = Parameter(gamma0)
116   Gamma = 150
117   b += Gamma*(PHI.Deriv()[0,0] - PHI.Deriv()[1,1])*(X.Deriv()[0,0] - X.Deriv()[1,1])*dx
118   b += Gamma*(PHI.Deriv()[1,0] - PHI.Deriv()[0,1])*(X.Deriv()[1,0] - X.Deriv()[0,1])*dx
119
120   # deformation  calculation
121   gfX = GridFunction(VEC)
122
123   def updateParams(v=False):
124       vol.Set( Integrate( surf_t ,mesh))
125       bc_x.Set((1/surf_0)*Integrate(bc_tx,mesh))
126       bc_y.Set((1/surf_0)*Integrate(bc_ty,mesh))
127       if(v):
128           print(vol.Get(), bc_x.Get(), bc_y.Get())
129 updateParams()
130
131 def increaseParams(k=2,v=False):
132       alpha.Set(alpha.Get()*k)
133       beta.Set(beta.Get()*k)
134       gamma.Set(gamma.Get()*k)
135       if(v):
136           print("alpha: ", alpha.Get(), "  beta: ", beta.Get(), "  gamma: ", gamma.Get())
137
138 def SolveDeformationEquation():
139       rhs = gfX.vec.CreateVector()
140       rhs.data = dJOmega.vec - b.mat * gfX.vec
141       update = gfX.vec.CreateVector()
142       update.data = b.mat.Inverse(VEC.FreeDofs()) * rhs
143       gfX.vec.data += update
144
145 gfset.Set((0,0))
146 mesh.SetDeformation(gfset)
147 scene.Redraw()
148
```

```
149 updateParams()
150 alpha0 = 1e-4
151 beta0 = 1e-0
152 gamma0 = 1e2
153 alpha.Set(alpha0)
154 beta.Set(beta0)
155 gamma.Set(gamma0)
156
157 a.Assemble()
158 solveStokes()
159
160 data = [[] for x in range(7)]
161
162 iter_max = 800
163 Jold = Integrate(calc_drag(gfu), mesh)
164
165 # try parts of loop
166 mesh.SetDeformation(gfset)
167 scene.Redraw()
168
169 c = 0
170
171 # input("Press enter to start optimization")
172 for i in range(0,iter_max):
173     mesh.SetDeformation(gfset)
174     scene.Redraw()
175     scene_state.Redraw()
176
177     if i%50 == 0:
178         print('drag at iteration', i, ': ', Jold)
179
180     titles = ["Energy Dissipation","Volume","bc_x","bc_y","scale","gfxnorm","gfxbndnorm"] # collecting data
181     data[0].append(Integrate(calc_drag(gfu),mesh))
182     data[1].append(vol.Get())
183     data[2].append(bc_x.Get())
184     data[3].append(bc_y.Get())
185
186     a.Assemble()
187     solveStokes()
188
189     b.Assemble()
190     dJOmega.Assemble()
191     SolveDeformationEquation()
192     updateParams()
193
194     Jnew = Integrate(calc_drag(gfu), mesh)
195     mesh.UnsetDeformation()
196
197     gfxnorm = Norm(gfX.vec)
198     gfxbndnorm = Integrate(sqrt(gfX**2),mesh,BND)
199     data[6].append(gfxbndnorm)
```

```
200     #scale = 0.1 / Norm(gfX.vec)
201     scale = 0.01 / gfxnorm
202     data[4].append(scale)
203     jdiff = abs(Jnew-Jold)
204     data[5].append(gfxnorm)
205
206
207     c += 1
208     #if(gfxnorm < 1e-5):
209     if(gfxbndnorm < 1e-7 and c > 50):
210         if alpha.Get() < 100:
211             increaseParams(5,True)
212             c = 0
213         else:
214             print("alpha too big")
215             break
216
217     gfsetOld = gfset
218     gfset.vec.data -= scale * gfX.vec
219     Jold = Jnew
220
221     Redraw(blocking=True)
222
223 Element_Iterator = mesh.Elements()
224 Element_Quality_Data = np.empty(Element_Iterator.stop)
225 deformation_vector = np.asarray(gfset.vec)
226
227 # Element Quality ratio eta with sidelengths a,b,c of element K
228 def eta_K(abc):
229     A, B, C = abc[0,:], abc[1,:], abc[2,:]
230     a, b, c = np.linalg.norm(A-C), np.linalg.norm(A-B), np.linalg.norm(B-C)
231     s = 1/2*(a+b+c) #half the circumference of element
232     A = np.sqrt(s*(s-a)*(s-b)*(s-c)) #Area calculation with Heron Theorem
233     r = 2*A / (a+b+c) # radius of inscribed circle
234     R = (a*b*c) / (4*A) # radius of circumscribed circle
235     eta = R / (2*r)
236     return eta
237
238 for elmnt in mesh.Elements():
239     el = NodeId(ELEMENT,elmnt.nr)
240     meshel = mesh[el]
241     abc = np.empty([3,2])
242     counter = 0
243
244     for vert in elmnt.vertices:
245         ve = NodeId(VERTEX, vert.nr)
246         meshvert = mesh[ve]
247         abc[counter] = np.asarray(meshvert.point) + deformation_vector[vert.nr]
248         counter += 1
249
250     counter = 0
```

```
251     Element_Quality_Data[elmnt.nr] = eta_K(abc)
```