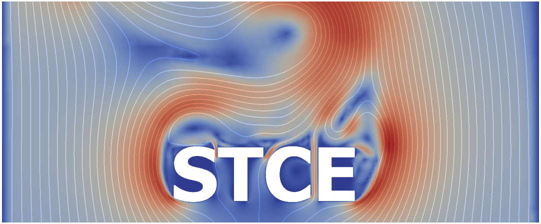


An Introduction to Discrete Adjoint Optimization with OpenFOAM

OpenFOAM Workshop 2019

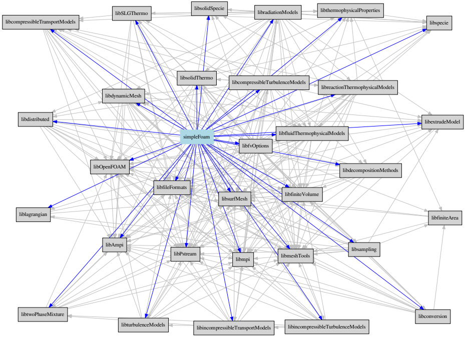
Dr. Markus Towara, feat. Andreas Pesch (RUB)

Duisburg, July 23th 2019

- this file: `https://stce.rwth-aachen.de/files/ofw19_slides.pdf`
- handout: `https://stce.rwth-aachen.de/files/ofw19_handout.pdf`
- Only need one of the following:
- Binaries: `https://stce.rwth-aachen.de/files/ofw19_binary.tar.gz`
- Docker image: `https://stce.rwth-aachen.de/files/ofw19_docker.tar.gz`
- VM: `https://stce.rwth-aachen.de/files/ofw19_vm.tar.gz`

OpenFOAM Softwarepackage

- Open-Source CFD software package (GPLv3)
- Finite volume discretization on 3D unstructured meshes
- Cell centered physical quantities (mostly)
- Highly complex C++ code, heavily relying on inheritance and templates
- Code: $\sim$ 1M LOC, 9k files
- Minimal external dependencies
- Parallelization using MPI

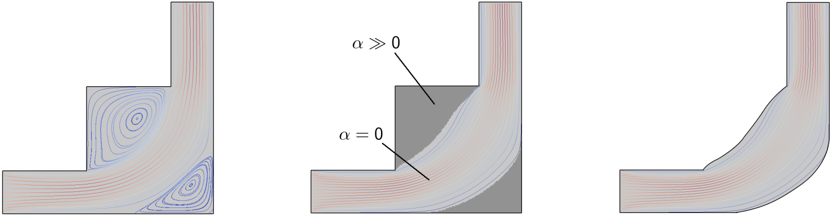


- Augment NS momentum equations by source term $\mathbf{u}\alpha$: [1, 2]

$$(\mathbf{u} \otimes \boldsymbol{\nabla})\,\mathbf{u} = \nu \boldsymbol{\nabla}^2 \mathbf{u} - \frac{1}{\rho}\boldsymbol{\nabla} p - \alpha \mathbf{u}$$

- Parameter α allows to penalize cells / regions of the geometry to redirect flow
- Penalty term can be interpreted as porosity, according to Darcy's law [3]

$$\frac{\Delta p}{\Delta x} = -\left(\frac{\mu}{\kappa}\right)\mathbf{u}$$



Initial design space | Optimized penalty field | Re-Parametrization

- Define cost function $\mathcal{J}$, e.g. total pressure loss between inlet and outlet:

$$\mathcal{J} = -\int_\Gamma \left(p + \frac{1}{2}\|\mathbf{u}\|^2\right) \mathbf{u} \cdot \mathbf{n} \,\mathrm{d}\Gamma$$

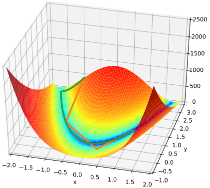
- Calculate sensitivity of the cost function w.r.t. parameters α_i

$$\frac{\mathrm{d}\mathcal{J}}{\mathrm{d}\alpha_i} = ???$$

- Calculate an updated porosity field α^{n+1}, e.g. using gradient descent:

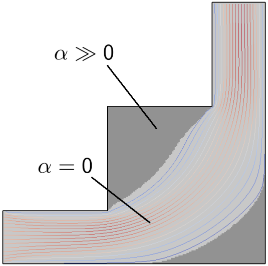
$$\alpha_i^{n+1} = \alpha_i^n - \lambda \cdot \frac{\mathrm{d}\mathcal{J}^n}{\mathrm{d}\alpha_i^n}, \quad \text{with constraints} \quad 0 \le \alpha_i \le \alpha_{\max}$$

- Loop until α converged...



Efficient optimization methods need gradients!

- Number of inputs $p = n$ to be optimized might be in the millions
- Calculating the gradient with finite differences extremely expensive
- Number of outputs usually $1 \leq m \ll n$
- Adjoint method allows to calculate the gradient with only m additional (augmented) function evaluations

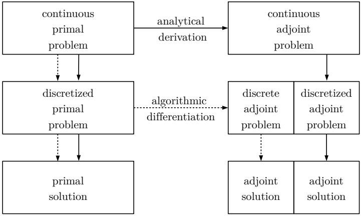


Continuous method:

- *Differentiate first, discretize later*
- Derive adjoint equations analytically
- Implement, discretize, and solve adjoint equations along primal

\+ Fast, physically interpretable

– Hard to derive, can be inconsistent to primal



Discrete method:

- *Discretize first, differentiate later*
- Use implementation to get the derivatives (Algorithmic Differentiation)

\+ Flexible, derivation automatic, sensitivities consistent to implementation

– Memory intensive, generally slower than continuous

- Consider multivariate function f mapping vector $\boldsymbol{x}$ to a scalar y:

$$f(\boldsymbol{x}) : \mathbb{R}^n \to \mathbb{R}$$

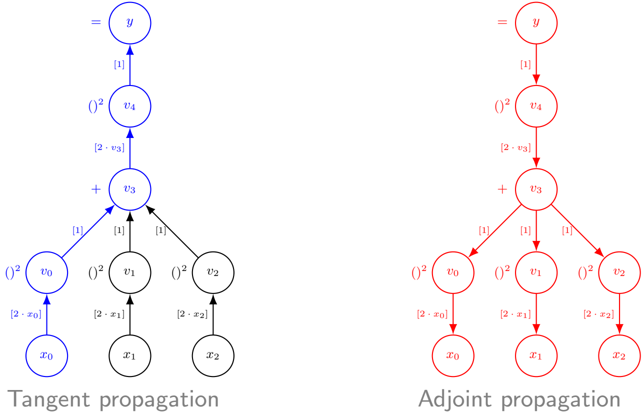
First Order Tangent Model	First Order Adjoint Model
$\dot{f}(\mathbf{x}, \dot{\mathbf{x}}) : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R} \times \mathbb{R}$	$\bar{f}(\mathbf{x}, \bar{y}) : \mathbb{R}^n \times \mathbb{R} \to \mathbb{R} \times \mathbb{R}^n$
$y = f(\boldsymbol{x})$	$y = f(\boldsymbol{x})$
$\dot{y} = \boldsymbol{\nabla} f(\boldsymbol{x}) \cdot \dot{\boldsymbol{x}}$	$\bar{\boldsymbol{x}} = \bar{\boldsymbol{x}} + \boldsymbol{\nabla} f(\boldsymbol{x})^T \cdot \bar{y}$

- Adjoint model is the obvious choice for high dimensional optimization problems

$$\min_{\boldsymbol{\alpha} \in \mathbb{R}^p} f(\boldsymbol{x}, \boldsymbol{\alpha}) \quad f(\boldsymbol{x}, \boldsymbol{\alpha}) : \mathbb{R}^n \times \mathbb{R}^p \to \mathbb{R}^m \quad \text{with} \quad p \gg m$$

- Second and higher derivatives can be obtained by nesting models

$$f(\mathbf{x}) = \left(x_0^2 + x_1^2 + x_2^2\right)^2 \qquad \underbrace{\nabla f(\mathbf{x})}_{\in \mathbf{R}^n} \equiv \frac{\partial y}{\partial \mathbf{x}} = \begin{pmatrix} (2x_0)\cdot(1)\cdot(2\cdot(x_0^2+x_1^2+x_2^2)) \\ (2x_1)\cdot(1)\cdot(2\cdot(x_0^2+x_1^2+x_2^2)) \\ (2x_2)\cdot(1)\cdot(2\cdot(x_0^2+x_1^2+x_2^2)) \end{pmatrix}$$



- Need to differentiate basic operation like $+, -, \sin, \exp, \ldots$
- Partial derivatives will be assembled using chain rule
- In C++ can be achieved by utilizing operator overloading, i.e. replacing intrinsic operations by custom ones
- Different tools available, e.g. `dco/c++`, ADOL-C, CoDiPack [4, 5, 6]
- Need to change datatypes of floating point values to custom datatpye

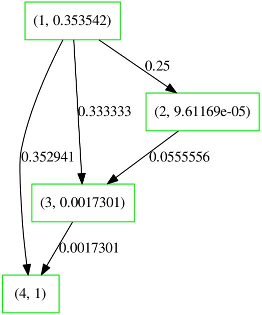


Figure: Internal `dco/c++` tape structure

OpenFOAM Adjoint Mode with dco/c++

- Idea: use central OpenFOAM typedef to replace all floating point values by custom AD data type [7]
- Should take care of bulk of AD work
- Minor manual adjustments needed, which accumulate over a large codebase

in `src/OpenFOAM/primitives/Scalar/doubleScalar/doubleScalar.h` replace:

```
namespace Foam{
  typedef double doubleScalar;
  ...
}
```

with:

```
#include "dco.hpp"
namespace Foam{
  typedef dco::ga1s<double>::type doubleScalar;
  ...
}
```

- Idea: use central OpenFOAM typedef to replace all floating point values by custom AD data type [7]
- Should take care of bulk of AD work
- Minor manual adjustments needed, which accumulate over a large codebase

in `src/OpenFOAM/primitives/Scalar/doubleScalar/doubleScalar.h` replace:

```
namespace Foam{
  typedef double doubleScalar;
  ...
}
```

with:

```
#include "dco.hpp"
namespace Foam{
  typedef dco::gt1s<double>::type doubleScalar;
  ...
}
```

Tangent vector mode can be implemented analogously (e.g. `dco::gt1v<double,16>::type`).

Features of Discrete Adjoint OpenFOAM

Groundworks:

- Typedef approach allows to differentiate the whole simulation code (Black-Box)
- For practical applications not feasible, further optimizations are needed
- A partial lists of features and methods enbabled by them are listed below

Algorithmic optimizations:

- Checkpointing [8]
- Reverse accumulation and Piggy-backing [9, 10]
- Symbolic differentiation of embedded linear solvers (SDLS) [11, 12]
- Adjoints of MPI parallelism by Adjoint-MPI [13, 12]

Case studies: [14]

- Topology optimization
- Parametric optimization
- Shape optimization

Currently working on CHT, see talk tomorrow.

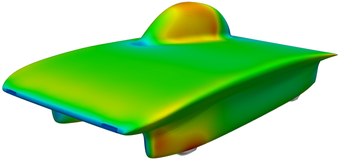


Figure: Shape sensitivity of Sonnenwagen w.r.t. (viscous) drag [15]

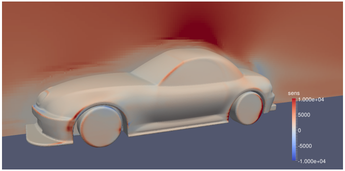


Figure: Shape sensitivity of touring car body [17]

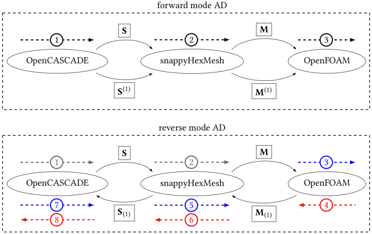


Figure: Differentiation of CAD toolchain [16]

- `wget https://stce.rwth-aachen.de/files/ofw19_docker.tar.gz`
- `tar -xzf ofw19_docker.tar.gz`
- import docker image and create container: `./create.sh`
- run and attach to container: `./run.sh`
- contents of `tutorial_data` will become the home directory of the docker container
- (discrete adjoint) OpenFOAM (v1812) installed in `/opt/discreteAdjointOpenFOAM-plus`

- Inspect environment with `env`
- Discrete Adjoint specific:
  - `DOF_COMPILER=Gcc`
  - `DOF_AD_OPTION=A1S`
  - `DOF_COMPILE_OPTION=Opt`
- Other `DOF_AD_OPTIONS` could be (not compiled in docker image):
  - `Passive` passive mode (double)
  - `A1S` adjoint mode
  - `T1S` tangent mode
  - `T2A1S` tangent over adjoint mode (2nd order)

- First we look at the implementation of a black box solver
- starting with `simpleFoam`
- due to memory demand not very practical, but a good starting point for other solvers
- you can follow along in `$OFW_DATA/adjointSimpleFoam/adjointSimpleFoam.C`

```
int main(int argc, char *argv[])
{
    #include "createFields.H"

    simpleControl simple(mesh);
    // run until end time reached / converged
    while (simple.loop())
    {
        // Pressure-velocity SIMPLE corrector
        #include "UEqn.H"
        #include "pEqn.H"

        turbulence->correct();
        runTime.write();
    }
    return 0;
}
```

Calculation of the Residual in OpenFOAM

Momentum Equation:

$$\nabla \cdot (\phi, \mathbf{U}) - \nabla \cdot (\nu \nabla \mathbf{U}) + \alpha \mathbf{U} = -\nabla \mathbf{p}$$

with mass flux through faces $\phi = \rho A \mathbf{U} \cdot \mathbf{n}$

`UEqn.H`:

```
fvVectorMatrix UEqn
(
    fvm::div(phi, U)
  - fvm::laplacian(nu, U)
  + fvm::Sp(alpha, U)
  ==
    fvOptions(U)
);
fvOptions.constrain(UEqn);
UEqn.relax();
fvVectorMatrix UEqnFull(UEqn == -fvc::grad(p));
```

Example Adjoint Solver, start with simpleFoam


```
int main(int argc, char *argv[]){
    ADmode::global_tape = ADmode::tape_t::create();
    ADmode::global_tape->register_variable(alpha[i],n);

    while (simple.loop()){
        #include "UEqn.H"
        #include "pEqn.H"
        turbulence->correct();
    }

    scalar J = 0;
    forAll(costFunctionPatches(),patchI)
        J += calcCost(patchI);

    dco::derivative(J) = 1.0;
    ADmode::global_tape->interpret_adjoint();

    // get adjoints, scale with cell volume, write to sens
    forAll(alpha,i){
        sens[i] = dco::derivative(alpha[i])/mesh.V()[i];
    }
}
```

```
int main(int argc, char *argv[]){
    ADmode::global_tape = ADmode::tape_t::create();
    ADmode::global_tape->register_variable(alpha[i],n);

    while (simple.loop()){
        #include "UEqn.H"
        #include "pEqn.H"
        turbulence->correct();
    }

    scalar J = 0;
    forAll(costFunctionPatches(),patchI)
        J += calcCost(patchI);

    dco::derivative(J) = 1.0;
    ADmode::global_tape->interpret_adjoint();

    // get adjoints, scale with cell volume, write to sens
    forAll(alpha,i){
        sens[i] = dco::derivative(alpha[i])/mesh.V()[i];
    }
}
```

```
int main(int argc, char *argv[]){
    ADmode::global_tape = ADmode::tape_t::create();
    ADmode::global_tape->register_variable(alpha[i],n);

    while (simple.loop()){
        #include "UEqn.H"
        #include "pEqn.H"
        turbulence->correct();
    }

    scalar J = 0;
    forAll(costFunctionPatches(),patchI)
        J += calcCost(patchI);

    dco::derivative(J) = 1.0;
    ADmode::global_tape->interpret_adjoint();

    // get adjoints, scale with cell volume, write to sens
    forAll(alpha,i){
        sens[i] = dco::derivative(alpha[i])/mesh.V()[i];
    }
}
```

```
int main(int argc, char *argv[]){
    ADmode::global_tape = ADmode::tape_t::create();
    ADmode::global_tape->register_variable(alpha[i],n);

    while (simple.loop()){
        #include "UEqn.H"
        #include "pEqn.H"
        turbulence->correct();
    }

    scalar J = 0;
    forAll(costFunctionPatches(),patchI)
        J += calcCost(patchI);

    dco::derivative(J) = 1.0;
    ADmode::global_tape->interpret_adjoint();

    // get adjoints, scale with cell volume, write to sens
    forAll(alpha,i){
        sens[i] = dco::derivative(alpha[i])/mesh.V()[i];
    }
}
```

- Symbolically differentiating Linear Solvers leaves GAP in Tape, needs to be filled during interpretation[1]
- The adjoint projections $\bar{A}$ and $\bar{\mathbf{b}}$ can be obtained by solving the additional equation system[2]:

$$A^T \cdot \bar{\mathbf{b}} = \bar{\mathbf{x}} \quad \Rightarrow \bar{\mathbf{b}}$$

- $\bar{A}$ can be obtained by calculating the outer product of $-\bar{\mathbf{b}}$ and $\mathbf{x}^T$:

$$\bar{A} = -\bar{\mathbf{b}} \cdot \mathbf{x}^T$$

[1] U. Naumann et al.: *Algorithmic Differentiation of Numerical Methods: First-Order Tangents and Adjoints for Solvers of Systems of Nonlinear Equations*, ACM TOMS, Vol. 41

[2] M. B. Giles: *Collected Matrix Derivative Results for Forward and Reverse Mode Algorithmic Differentiation*

```
SDLS yes;

solvers{
  "(.*)" {
    solver          smoothSolver;
    smoother        symGaussSeidel;
    tolerance       1e-05;
    relTol          0;
    SDLS            $SDLS;
  }
  "(p|pReverse|Phi)" {
    solver          GAMG;
    tolerance       1e-06;
    relTol          0;
    smoother        DIC;
    SDLS            $SDLS;
  }
}

SIMPLE {
  nNonOrthogonalCorrectors 0;
  consistent        yes;
  costFunctionPatches (inlet outlet);
  costFunction "pressureLoss";
}
```

Checkpointing

- Trade memory demand for run time
- Only adjoin one time step at a time, then restore primal from an earlier time step and recalculate, record and adjoin next iteration step.
- Online Checkpointing using `Revolve` or `Equidistant`
- `cd $OFW_DATA/adjointSimpleCheckpointingFoam`
- `cd pitzDaily`
- inspect `system/checkpointingDict`

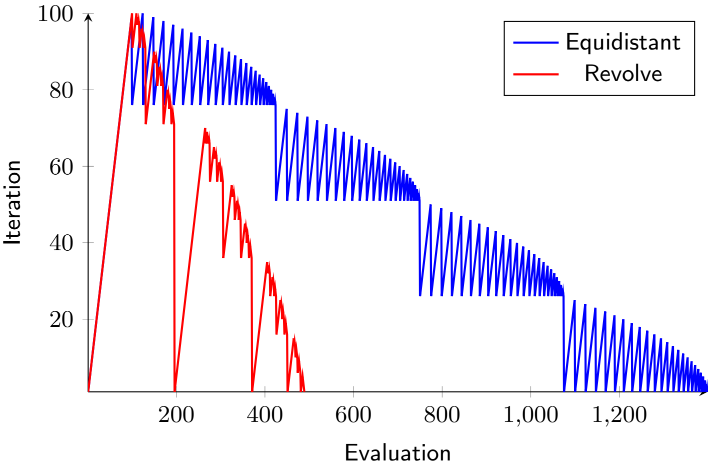
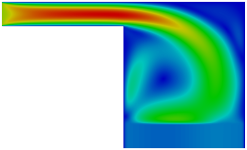
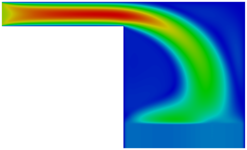


Figure: Revolve and equidistant for 100 iteration steps and 4 checkpoints.

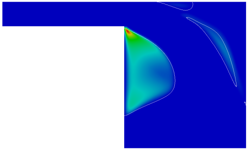
- For steady state cases one can utilize reverse accumulation or piggy backing
- Recording of single iteration step is repeatedly adjoined, forming a fixed point iteration yielding the correct adjoints
- Comparable to continuous adjoint, where adjoints are propagated forward alongside the primal
- Solver: `$OFW_DATA/piggyOptSimpleFoam`
- Case: `$OFW_DATA/piggyOptSimpleFoam/filter_case` with porosity at outlet, reconstructed from first Othmer paper [2].



Initial flow field



Optimized flow field



Topology penalty field

Lets implement our own cost function

- Solver: `$OFW_DATA/flowUniformity`
- Case: `$OFW_DATA/flowUniformity/flow_uniformity_case`
- Combination of flow uniformity and pressure loss function

```
scalar Jp = CostFunction(mesh).eval();
Foam::wordList outlets(2);
outlets[0]="outlet0";
outlets[1]="outlet1";

std::vector<scalar> meanMagU(outlets.size(),0.0);
forAll(outlets,cI)
{
  label patchI = mesh.boundaryMesh().findPatchID(outlets[cI] );
  fvPatch& patch = mesh.boundary()[patchI];
  meanMagU[cI] = gAverage(phi.boundaryField()[patchI]/patch.magSf());
}
scalar Jv = pow(meanMagU[0]-meanMagU[1],scalar(2.0));

scalar J = Jp + 0.0002 * Jv;
dco::derivative(J) = 1.0;
```

Result

- Secondary design goal pressure loss necessary
- else big sponge is a feasible solution

Initial flow field

Optimized flow field

Topology penalty field

- more sophisticated optimization methods should be explored
- multiple objectives can be evaluated with same tape (adjoint vector mode or sequential)
- can also use external optimizers, e.g. ceres, pyOpt

Discrete adjoint workflow:

- Use individual mesh point locations $\mathbf{P}$ as parameters
- Interpolate adjoint sensitivities $\bar{\mathbf{P}}$ of points to boundary face centers $\Rightarrow \bar{\mathbf{P}}_F$
- Take scalar product with face normal $\mathbf{n}_F$, divide by face area A_F: $s = \frac{\bar{\mathbf{P}}_F \cdot \mathbf{n}_F}{A_F}$

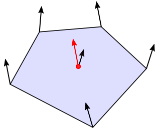


Figure: Point to face midpoint interpolation of sensitivity vectors

Continuous adjoint workflow:

- Calculate adjoint velocities $\mathbf{v}$ and pressure q using continuous adjoint NS[3]
- Surface sensitivity: $\frac{\partial \mathcal{J}}{\partial \beta} = -A\nu(\mathbf{n} \cdot \nabla)\mathbf{u}_t \cdot (\mathbf{n} \cdot \nabla)\mathbf{v}_t$

[3] calculated using OpenFOAM `adjointShapeOptimization` (sic!) solver modified to obtain shape adjoints according to [2]

- Consider laminar flow past cylinder at $Re = 2$ and $Re = 20$
- *Structured* non-orthogonal 2D mesh
- Compare discrete and adjoint sensitivities w.r.t. surface drag, obtained by the significantly different approaches outlined before

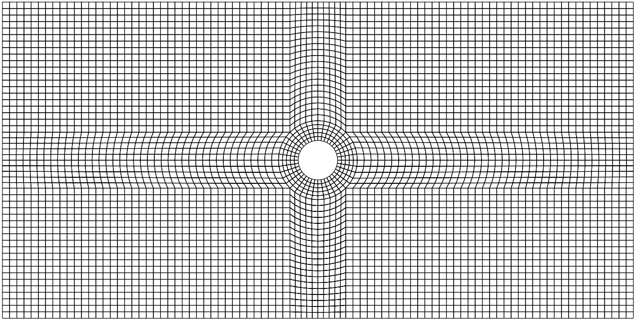
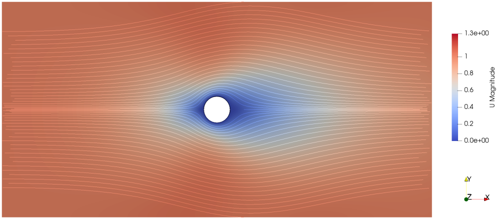
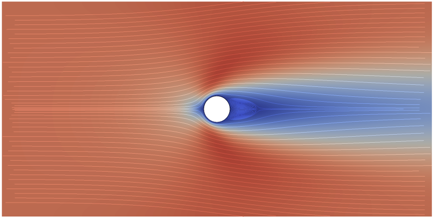


Figure: Structured non-cartesian mesh around cylinder

Re=2



Re=20



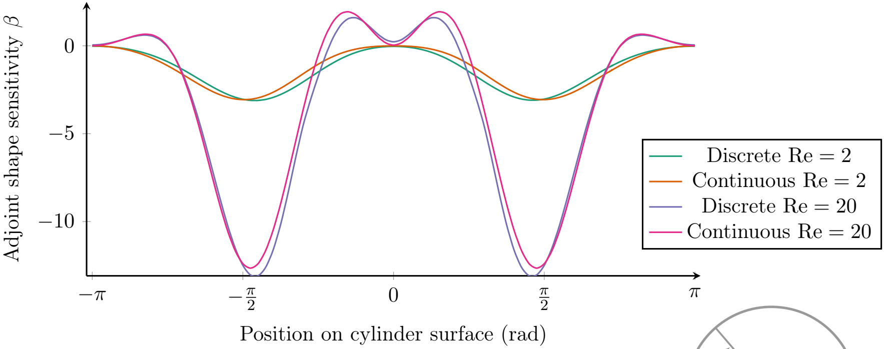
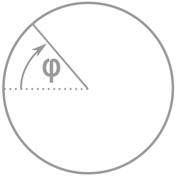


Figure: Surface sensitivity over angle



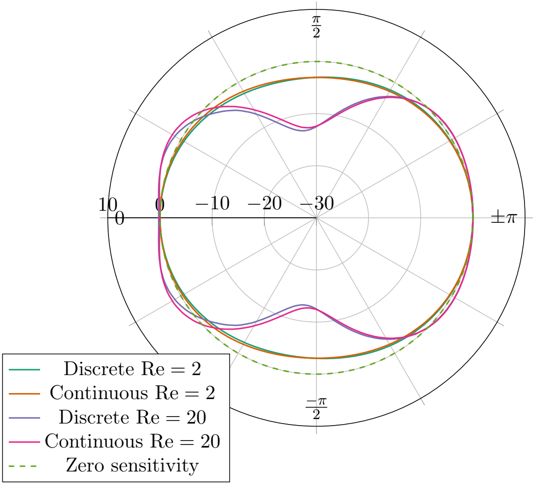
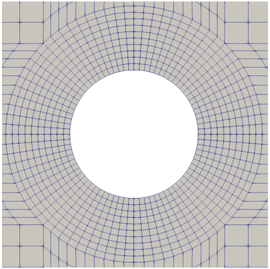
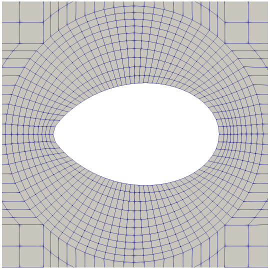


Figure: Surface sensitivity on cylinder surface in polar coordinates

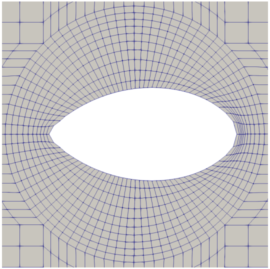
- Case: `$OFW_DATA/piggyShapeSimpleFoam/cylinderMirror`
- based on standard OpenFOAM tutorial case `cylinder`
- remove mirror plane by `mirrorMesh`
- `adjointMoveMesh` moves mesh points
- `adjointShapeOptimizationFoam` calculates sensitivities (with or without volume constraint).



Initial mesh



Optimization step 100



Optimization step 200

Thats all folks!

- Questions? `towara@stce.rwth-aachen.de`
- Full source access available `stce.rwth-aachen.de/foam`
- v1906 merge to appear soon

T. Borrvall and J. Petersson.
Topology optimization of fluids in Stokes flow.
International Journal for Numerical Methods in Fluids, 41(1):77–107, 2003.

C. Othmer.
A continuous adjoint formulation for the computation of topological and surface sensitivities of ducted flows.
International Journal for Numerical Methods in Fluids, 58(8):861–877, 2008.

S. Whitaker.
Flow in porous media i: A theoretical derivation of Darcy's law.
Transport in Porous Media, 1(1):3–25, 1986.

K. Leppkes, J. Lotz, and U. Naumann.
Derivative Code by Overloading in C++ (dco/c++): Introduction and Summary of Features.
Technical Report AIB-2016-08, RWTH Aachen University, September 2016.

A. Walther and A. Griewank.
Getting started with Adol-C.
U. Naumann and O. Schenk, Combinatorial Scientific Computing, Chapman-Hall CRC Computational Science, pages 181–202, 2012.

M. Sagebaum, T. Albring, and N. R. Gauger.
High-performance derivative computations using CoDiPack.
arXiv preprint arXiv:1709.07229, 2017.

M. Towara and U. Naumann.
A discrete adjoint model for OpenFOAM.
Procedia Computer Science, 18(0):429 – 438, 2013.
2013 International Conference on Computational Science.

A. Griewank and A. Walther.
Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation.
SIAM, 2008.

B. Christianson.
Reverse accumulation and attractive fixed points.
Optimization Methods and Software, 3(4):311–326, 1994.

Andreas Griewank and Christèle Faure.
Piggyback differentiation and optimization.
Large-Scale PDE-Constrained Optimization, pages 148–164, 2003.

M. B. Giles.
Collected matrix derivative results for forward and reverse mode algorithmic differentiation.
In *Advances in Automatic Differentiation*, pages 35–44. Springer, 2008.

M. Towara, M. Schanen, and U. Naumann.
MPI-parallel discrete adjoint OpenFOAM.
Procedia Computer Science, 51:19 – 28, 2015.
2015 International Conference On Computational Science.

M. Schanen.
Semantics Driven Adjoints of the Message Passing Interface.
Dissertation, RWTH Aachen University, 2014.

Markus Towara.
Discrete Adjoint Optimization with OpenFOAM.
Dissertation, RWTH Aachen University, 2019.

L. Moltrecht.
Adjoint-Based Aerodynamic Optimization of a Solar Vehicle Concept.
Master Thesis, RWTH Aachen University, 2018.

S. Gezgin.
Algorithmic Differentiation of a CAD Geometry Kernel and a Mesh Generation Tool.
Master Thesis, RWTH Aachen University, 2016.

A. Pesch.
Discrete Adjoint Optimization of a Rear Wing.
Master Thesis, RWTH Aachen University, 2016.