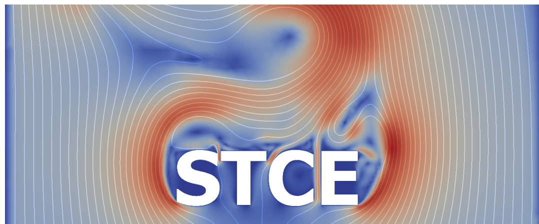


Discrete Adjoint Sensitivities in OpenFOAM

Markus Towara, Arindam Sen, Uwe Naumann
Software and Tools for Computational Engineering Science
RWTH Aachen University

11th OpenFOAM® Workshop, Guimarães June 2016

Outline

Introduction

Features of discrete adjoint OpenFOAM

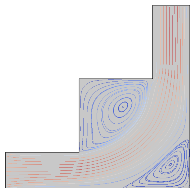
Applications

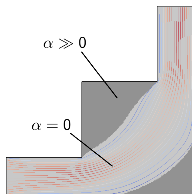
Why OpenFOAM?

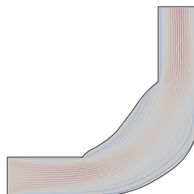
- Open-Source (GPLv3) CFD solver
- includes tools for meshing, pre-, post-processing
- rising adoption in industry and academia due to lack of licence costs → well suited for parallel architectures
- minimal dependence on external libraries
- comprehensible C++

Motivation: Topology Optimization

no optimization | added "material" | reconstruction







Add penalty term α to Navier-Stokes equation[1]

$$(\mathbf{v} \cdot \nabla)\,\mathbf{v} = \nu \nabla^2 \mathbf{v} - \nabla p - \alpha \mathbf{v}$$

[1] Othmer, C., E. de Villiers, and H.G. Weller: *Implementation of a continuous adjoint for topology optimization of ducted flows*

How to find appropriate α

RWTH AACHEN UNIVERSITY STCE

- Define Cost Function J, e.g. total pressure loss between inlet and outlet:

$$J = \int_\Gamma p + \frac{1}{2} v_n^2 \, \mathrm{d}\Gamma$$

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

$$\frac{\partial J}{\partial \alpha_i} = ???$$

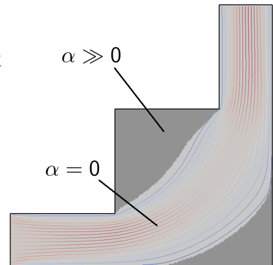
- Calculate updated porosity field α^{n+1}, e.g.:

$$\alpha_i^{n+1} = \alpha_i^n - \lambda \cdot \frac{\partial J^n}{\partial \alpha_i^n}, \quad \text{while insuring} \quad 0 < \alpha_i < \alpha_{max}$$

- Loop until α converged...

We need gradients!

- number of inputs n to be optimized might be in the millions
- calculating the gradient with finite differences extremely expensive
- number of outputs usually $m = O(1)$
- adjoint method allows to calculate the gradient with only m additional function evaluations



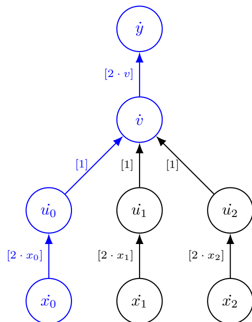
How to get an adjoint?

Continuous method:

- derive adjoint equations by hand → lots of variatonal calculus
- implement, discretize and solve adjoint equations along primal
- basic implementation available in `adjointShapeOptimizationFoam`
- \+ fast, physically interpretable
- − hard to derive, can be inconsistent to primal

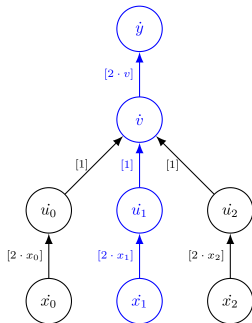
Discrete method:

- use implementation to get the derivatives (Algorithmic Differentiation)
- either use source code transformation or operator overloading
- \+ flexible, derivation automatic, consistent to implementation
- − memory intensive, generally slower than continuous



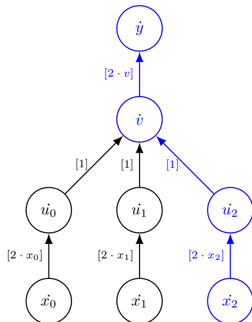
$$f(x) = \left(\sum_{i=0}^{n-1} x_i^2 \right)^2$$

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



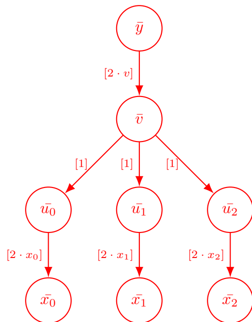
$$f(x) = \left(\sum_{i=0}^{n-1} x_i^2 \right)^2$$

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



$$f(x) = \left(\sum_{i=0}^{n-1} x_i^2 \right)^2$$

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



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

Problem:
Reversal of control flow in the derivative calculation, might depend on values from the primal which are already overwritten $\rightarrow$ need to store those values

Algorithmic Differentiation by Operator Overloading

- Need to differentiate basic operation like +,-,sin,exp ...
- Rest will be taken care of by chain rule
- Can be achieved by using Operator Overloading, i.e. replacing intrinsic operations by custom ones
- Need to change datatypes of floating point values to custom datatpye
- Different Tools available, e.g. dco/c++, ADOL-C, CoDiPack

Example Adjoint Solver, start with simpleFoam

RWTH AACHEN UNIVERSITY STCE

```
1  int main(int argc, char *argv[])
2  {
3      #include "createFields.H"
4
5      simpleControl simple(mesh);
6      // run until end time reached / converged
7      while (simple.loop())
8      {
9          // Pressure-velocity SIMPLE corrector
10         #include "UEqn.H"
11         #include "pEqn.H"
12
13         turbulence->correct();
14         runTime.write();
15     }
16     return 0;
17 }
```

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

```
1  int main(int argc, char *argv[]){
2      dco::ga1s<double>::global_tape = dco::ga1s<double>::tape_t::create();
3      dco::ga1s<double>::global_tape->register_variable(alpha[i],n);
4
5      while (simple.loop()){
6          #include "UEqn.H"
7          #include "pEqn.H"
8          turbulence->correct();
9      }
10
11     scalar J = 0;
12     forAll(costFunctionPatches(),patchI)
13         J += calcCost(patchI);
14
15     dco::derivative(J) = 1.0;
16     dco::ga1s<double>::global_tape->interpret_adjoint();
17
18     forAll(alpha,i) // get adjoint sensitivitys, scale with cell volume, write to sens
19         sens[i] = dco::derivative(alpha[i])/mesh.V()[i];
20 }
```

Example Adjoint Solver, start with simpleFoam

RWTH AACHEN UNIVERSITY STCE

```
1  int main(int argc, char *argv[]){
2      dco::ga1s<double>::global_tape = dco::ga1s<double>::tape_t::create();
3      dco::ga1s<double>::global_tape->register_variable(alpha[i],n);
4
5      while (simple.loop()){
6          #include "UEqn.H"
7          #include "pEqn.H"
8          turbulence->correct();
9      }
10
11     scalar J = 0;
12     forAll(costFunctionPatches(),patchI)
13         J += calcCost(patchI);
14
15     dco::derivative(J) = 1.0;
16     dco::ga1s<double>::global_tape->interpret_adjoint();
17
18     forAll(alpha,i) // get adjoint sensitivitys, scale with cell volume, write to sens
19         sens[i] = dco::derivative(alpha[i])/mesh.V()[i];
20 }
```

Example Adjoint Solver, start with simpleFoam

```
1  int main(int argc, char *argv[]){
2      dco::ga1s<double>::global_tape = dco::ga1s<double>::tape_t::create();
3      dco::ga1s<double>::global_tape->register_variable(alpha[i],n);
4
5      while (simple.loop()){
6          #include "UEqn.H"
7          #include "pEqn.H"
8          turbulence->correct();
9      }
10
11     scalar J = 0;
12     forAll(costFunctionPatches(),patchI)
13         J += calcCost(patchI);
14
15     dco::derivative(J) = 1.0;
16     dco::ga1s<double>::global_tape->interpret_adjoint();
17
18     forAll(alpha,i) // get adjoint sensitivitys, scale with cell volume, write to sens
19         sens[i] = dco::derivative(alpha[i])/mesh.V()[i];
20 }
```

Example Adjoint Solver, start with simpleFoam

```
1  int main(int argc, char *argv[]){
2      dco::ga1s<double>::global_tape = dco::ga1s<double>::tape_t::create();
3      dco::ga1s<double>::global_tape->register_variable(alpha[i],n);
4
5      while (simple.loop()){
6          #include "UEqn.H"
7          #include "pEqn.H"
8          turbulence->correct();
9      }
10
11     scalar J = 0;
12     forAll(costFunctionPatches(),patchI)
13         J += calcCost(patchI);
14
15     dco::derivative(J) = 1.0;
16     dco::ga1s<double>::global_tape->interpret_adjoint();
17
18     forAll(alpha,i) // get adjoint sensitivitys, scale with cell volume, write to sens
19         sens[i] = dco::derivative(alpha[i])/mesh.V()[i];
20 }
```

Features

- Black-Box derivatives
  - differentiate through whole iteration history

Features

- Black-Box derivatives
  - differentiate through whole iteration history
- Continuously Differentiated Linear Solvers
  - significantly lowers memory consumption
  - consumption independent of number of inner iterations

Features

- Black-Box derivatives
  - differentiate through whole iteration history
- Continuously Differentiated Linear Solvers
  - significantly lowers memory consumption
  - consumption independent of number of inner iterations
- Checkpointing
  - reduce memory footprint by trading memory for run-time
  - binomial checkpointing (REVOLVE) implemented

Features

- Black-Box derivatives
  - differentiate through whole iteration history
- Continuously Differentiated Linear Solvers
  - significantly lowers memory consumption
  - consumption independent of number of inner iterations
- Checkpointing
  - reduce memory footprint by trading memory for run-time
  - binomial checkpointing (REVOLVE) implemented
- Reverse Accumulation
  - repeatedly differentiate last iteration step to obtain derivatives
  - (fix point iteration)

Features

- Black-Box derivatives
  - differentiate through whole iteration history
- Continuously Differentiated Linear Solvers
  - significantly lowers memory consumption
  - consumption independent of number of inner iterations
- Checkpointing
  - reduce memory footprint by trading memory for run-time
  - binomial checkpointing (REVOLVE) implemented
- Reverse Accumulation
  - repeatedly differentiate last iteration step to obtain derivatives
  - (fix point iteration)
- Piggy-Backing
  - as Reverse Accumulation but apply gradient optimization after each iteration
  - One Shot Optimization

Features

- Black-Box derivatives
  - differentiate through whole iteration history
- Continuously Differentiated Linear Solvers
  - significantly lowers memory consumption
  - consumption independent of number of inner iterations
- Checkpointing
  - reduce memory footprint by trading memory for run-time
  - binomial checkpointing (REVOLVE) implemented
- Reverse Accumulation
  - repeatedly differentiate last iteration step to obtain derivatives
  - (fix point iteration)
- Piggy-Backing
  - as Reverse Accumulation but apply gradient optimization after each iteration
  - One Shot Optimization
- AMPI (Adjoint MPI)
  - retain parallelism of the primal also during adjoint
  - ease memory requirements by distributing to several machines

Test case: DrivAer

- Reynolds Number (Re) 30000, RANS with k-Ω-SST Turbulence Model
- Adjoint Solver: discreteAdjoint version of simpleFoam
- Sensitivities, $\frac{\partial J}{\partial \beta} = ?$
- β is the design variable, normal motion of surface nodes.

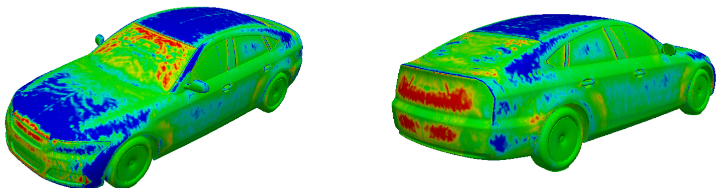


Figure: Drag sensitivities of DrivAer.
To reduce drag, Blue: push in (fluid to solid), Red: pull out (solid to fluid)

Aeroacoustic Noise

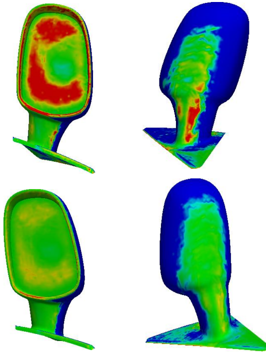


Figure: Noise and Drag sensitivities on the mirror surface

Approximate Cost function:

$$J = \int_{CV} \nu_t^2 dV \tag{1}$$

Summary and Outlook

- Discrete Adjoints available for whole OpenFOAM core package
- creation of new solvers pretty straightforward
- MPI Parallelism retained
- Open-Source, please contact us for access!
- visit: `https://www.stce.rwth-aachen.de/foam`
- contact: `towara@stce.rwth-aachen.de`

Acknowledgements

Parts of this work have been conducted within the **About Flow** project on "Adjoint-based optimisation of industrial and unsteady flows".

http://aboutflow.sems.qmul.ac.uk

About Flow has received funding from the European Union's Seventh Framework Programme for research, technological development and demonstration under Grant Agreement No. 317006.

National Technical University of Athens

Ownership of the OpenFOAM® trademark by ESI Group is acknowledged. This offering is not endorsed by the owner of the trademark.

Thank you!

Questions?