Current Developments and Applications related to the Discrete Adjoint Solver in SU2

Tim Albring, Max Sagebaum, Ole Burghardt,
Lisa Kusch, Nicolas R. Gauger

Chair for Scientific Computing
TU Kaiserslautern

Scientific Computing

SU2
The Open-Source CFD Code

December 18, 2017

Basics of Algorithmic Differentiation (AD)

- AD exploits the fact that **any computer program** that evaluates a function $z = f(x)$ is merely a sequence of statements (expressions):

$$z = f(x) = h_n(h_{n-1}(\dots h_1(x)))$$

- In the **Forward Mode** of AD we traverse the chain rule from right to left (*How does an infinitely small change in the input values affect the output?*):

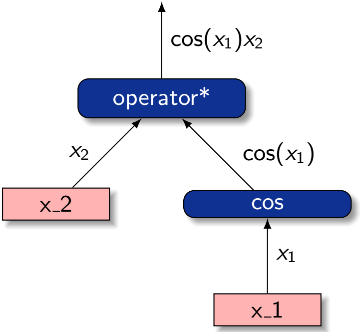
$$\dot{z} := \frac{df}{dx} \cdot \dot{x} = \frac{dh_n}{dh_{n-1}} \cdot \frac{dh_{n-1}}{dh_{n-2}} \dots \frac{dh_1}{dx} \cdot \dot{x}$$

- For the **Reverse Mode** of AD the chain rule is applied from left to right (*How sensitive are the output values to a change in the input values?*):

$$\bar{x} := \left(\frac{df}{dx}\right)^T \cdot \bar{z} = \left(\frac{dh_1}{dx}\right)^T \cdot \left(\frac{dh_2}{dh_1}\right)^T \dots \left(\frac{dh_n}{dh_{n-1}}\right)^T \cdot \bar{z}$$

- Derivatives of expressions can be efficiently evaluated using the Expression Template technique.

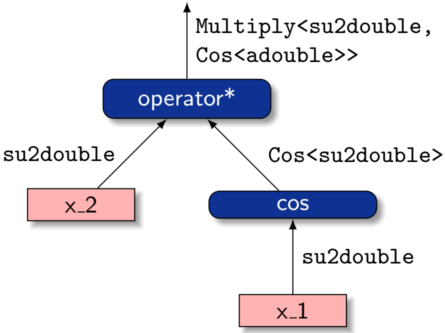
Expression Templates in CoDi



Computational graph for the expression $h_1 = \cos(x_1)x_2$.

- Each statement consists of a sequence of elementary operations $(+, *, \sin, \cos$ etc.$)$ that can be **easily differentiated**.
- Idea: create a internal representation of each expression at **compile-time**.

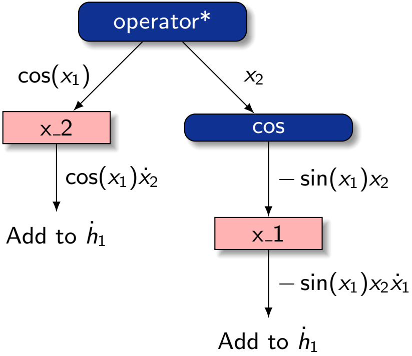
Expression Templates in CoDi



Compile-time representation of types for the expression $h_1 = \cos(x_1)x_2$.

- Each statement consists of a sequence of elementary operations $(+, *, \sin, \cos$ etc.$)$ that can be **easily differentiated**.
- Idea: create a internal representation of each expression at **compile-time**.
- Overload each operation to return an **object** representing this operation and its arguments.

Expression Templates in CoDi



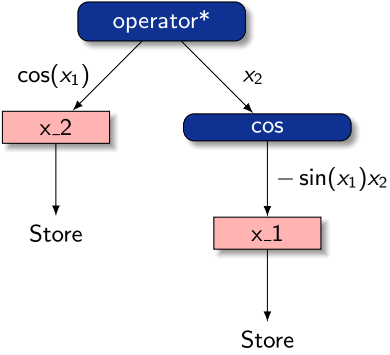
Run-time traversal for the expression $h_1 = \cos(x_1)x_2$.

- Each statement consists of a sequence of elementary operations $(+, *, \sin, \cos$ etc.$)$ that can be **easily differentiated**.
- Idea: create a internal representation of each expression at **compile-time**.
- Overload each operation to return an **object** representing this operation and its arguments.
- Expression object can be traversed at **run-time** to accumulate the gradients.

For the Forward mode the gradients are immediately constructed:

$$\dot{h}_1 = \frac{\partial h_1}{\partial x_1}\dot{x}_1 + \frac{\partial h_1}{\partial x_2}\dot{x}_2 = -\sin(x_1)x_2\dot{x}_1 + \cos(x_1)\dot{x}_2$$

Expression Templates in CoDi



Run-time traversal for the expression $h_1 = \cos(x_1)x_2$.

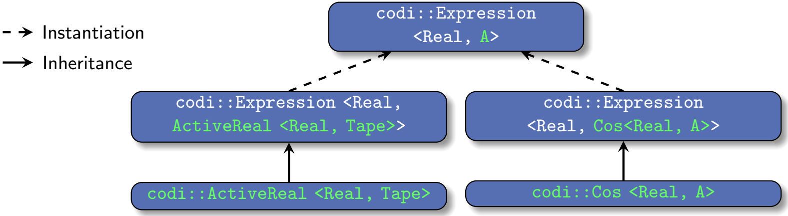
- Each statement consists of a sequence of elementary operations $(+, *, \sin, \cos$ etc.) that can be **easily differentiated**.
- Idea: create a internal representation of each expression at **compile-time**.
- Overload each operation to return an **object** representing this operation and its arguments.
- Expression object can be traversed at **run-time** to accumulate the gradients.

Gradients are accumulated in a second (reverse) sweep using stored information:

$$\bar{x}_1 = \bar{x}_1 + \frac{\partial h_1}{\partial x_1}\bar{h}_1 = \bar{x}_1 - \sin(x_1)x_2\bar{h}_1$$

$$\bar{x}_2 = \bar{x}_2 + \frac{\partial h_1}{\partial x_2}\bar{h}_1 = \bar{x}_2 + \cos(x_1)\bar{h}_1$$

Expressions and Active Real Definition using CRTP



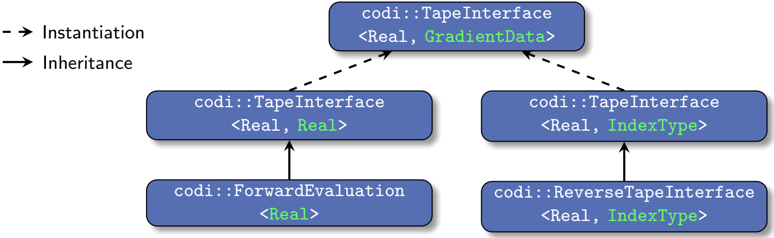
Curiously Recurring Template Pattern (CRTP) enables **static polymorphism**. Each of the derived expressions implements a `calcGradient()` routine that computes its (partial) derivative and calls the `calcGradient()` routine of its arguments.

```
template <typename Real, class A>
class Expression{
  inline const A& cast() const {
    return static_cast<const A&>(*this);
  }
  inline void calcGradient(Real& gradient,
                const Real& multiplier) const {
    cast().calcGradient(gradient, multiplier);
  }
}
```

```
template<typename Real, class A>
struct Cos :
public Expression<Real, Cos<Real, A>>{
  const &A a_;
  inline void calcGradient(Real& gradient,
               const Real& multiplier) const {
    a_.calcGradient(gradient,
          -sin(a_.getValue())*multiplier);
  }
}
```

The overridden function in the derived class is selected at compile time.

(Simplified) Tape Interface Definition



Common abstract interface for forward and reverse mode. It defines functions to signal the tape implementation when

- an ActiveReal is **constructed** or **destroyed**
- the **assignment operator (=)** of the ActiveReal with active RHS (Expression) is called (triggers calcGradient() of this expression)
- an ActiveReal is **input** of an expression (calcGradient() of ActiveReal, terminates the gradient computation of this expression)

su2double is actually (by default) one of the following types:
Reverse mode: ActiveReal<JacobiTape<ChunkTapeTypes<double, LinearIndexHandler<int>>>>

Forward mode: ActiveReal<ForwardEvaluation<double>>

CoDiPack - Code Differentiation Package for C/C++

Why yet another AD tool ?

- Compile-time construction of statement objects using Expression Templates
  → yields **high performance** and possibility to analyze source code
- Flexible **template-based** implementation
- Distinct interface between the Expression Template implementation and the tape implementation
  → allows **different taping methods** (primal value taping, Jacobi taping, memory handling using chunks, preallocated memory etc)
- Available as Open-source under GPL3 on Github (https://github.com/SciCompKL/CoDiPack)
- Extensive documentation and tutorials (more will be added in the future)
- Automatic self-testing (also on TravisCI)
- Header-only

MPI and AD: First Challenge

There exists a huge variety of AD tools, e.g.

Operator Overloading AD

- CoDiPack
- ADOL-c
- dco/c++
- Adept
- FADBAD
- Sacado
- etc.

Source Transformation AD

- Tapenade
- OpenAD
- ADIC
- etc.

All of them have different approaches on how to store data.

MPI and AD: Second Challenge

The MPI standard is comprehensive ...

Functions

Bsend, Ibsend, Imrecv, Irecv, Irsend, Isend, Issend, Mrecv, Recv, Rsend, Send, Sendrecv, Ssend, Allgather, Allgatherv, Allreduce_global, Alltoall, Alltoallv, Bcast_wrap, Gather, Gatherv, Iallgather, Iallgatherv, Iallreduce_global, Ialltoall, Ialltoallv, Ibcast_wrap, Igather, Igatherv, Ireduce_global, Iscatter, Iscatterv, Reduce_global, Scatter, Scatterv, etc.

Standards

- MPI 1.*: 129 Functions
- MPI 2.*: 183 functions
- MPI 3.*: 109 functions
- Total: 421 functions

MPI and AD: Second Challenge cont'd

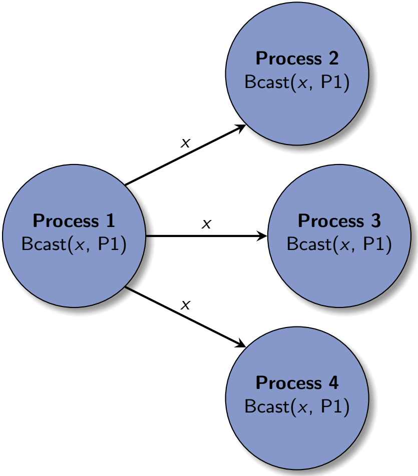
Concepts (Overview)

- Send buffer
- Recv buffer
- Inplace buffers
- Communicators (intra and inter)
- Collective (multiple ranks)
- Variable size per rank
- Asynchronous
- Reduction operation
- Custom data types
- Message fitting
- Preinitialization

All of these concepts must be handled to work with AD datatypes.

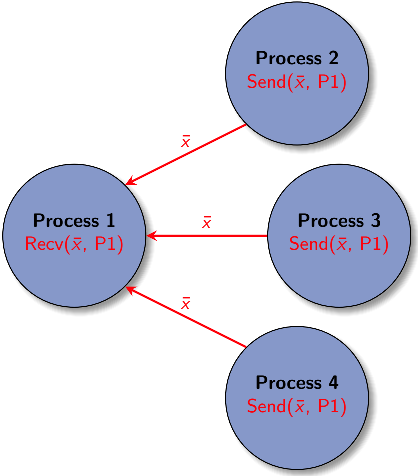
Reverse Mode of AD and MPI

Broadcast Example



Reverse Mode of AD and MPI

Broadcast Example



Message Differentiation Package

Features:

- A full forward of AMPI_ to MPI_
- 80% (340/421) coverage of the full MPI standard up to now
  - MPI 1.* 90% (117/129)
  - MPI 2.* 83% (153/183)
  - MPI 3.* 64% (70/109)
- Uses a code generator to avoid duplicated code for common concepts (improves maintainability)
- Header-only library
- Available as open-source on Github: https://github.com/scicompkl/medipack

What does that mean for SU2:

- Integration almost finished (automatically downloaded with the `preconfigure.py` script)
- All MPI calls can be replaced with `SU2_MPI::` wrapper calls
- Future-proof: possibility to easily handle e.g. higher-order derivatives and/or new MPI communication concepts

Abstract Fixed-Point Formulation for Multi-Disciplinary Design

- $\beta \in \mathbb{R}^p$: design vector
- $U \in \mathbb{R}^n$: state vector
- $X \in \mathbb{R}^m$: computational mesh
- $\mathcal{M}(\beta) = X$: mesh deformation equation
- $J(U, X)$: objective function
- $\mathcal{R}(U, X) = 0$: discretized state equation

Note: $\mathcal{R}$ or rather $\mathcal{G}$ contain **everything*** implemented in the code.
Has been applied in SU2 so far to

- Coupled problems (FSI and CHT),
- Turbomachinery problems,
- Aeroacoustics,
- Harmonic Balance,
- etc.

*at least by default

$$\begin{aligned} \min_{\beta} \quad & J(U(\beta), X(\beta)) & \\ \text{s.t.} \quad & \mathcal{R}(U(\beta), X(\beta)) &= 0 \\ & \mathcal{M}(\beta) &= X \end{aligned}$$

Assuming $\mathcal{R}(U, X) = 0$ is solved by a fixed-point iteration: $\mathcal{G}(U^*, X) = U^* \Leftrightarrow \mathcal{R}(U^*, X) = 0$

$$\begin{aligned} \min_{\beta} \quad & J(U(\beta), X(\beta)) & \\ \text{s.t.} \quad & \mathcal{G}(U(\beta), X(\beta)) &= U \\ & \mathcal{M}(\beta) &= X \end{aligned}$$

In case of Newton-type solver: $\mathcal{G}(U, X) := U - \mathcal{P}(U, X)\mathcal{R}(U, X)$, where $\mathcal{P} \approx (\partial\mathcal{R}/\partial U)^{-1}$.

The Discrete Adjoint Solver

Using the method of Lagrangian multiplier we define the **Lagrangian function** as:

$$\mathcal{L}(\beta, U, X, \bar{U}, \bar{X}) = \underbrace{J(U,X) + \bar{U}^T(\mathcal{G}(U,X)}_{=:\mathcal{N},\ \textit{Shifted Lagrangian}} - U) + \bar{X}^T(\mathcal{M}(\beta) - X)$$

KKT conditions yield equations for adjoints $\bar{U}, \bar{X}$ and sensitivity vector $d\mathcal{L}/d\beta$:

$$\begin{aligned}
\bar{U} &= \frac{\partial}{\partial U} J(U,X) + \frac{\partial}{\partial U}\mathcal{G}^T(U,X)\bar{U} \\
&= \frac{\partial}{\partial U}\mathcal{N}^T(U,\bar{U},X) \qquad \textbf{Adjoint equation} \\
\bar{X} &= \frac{\partial}{\partial X} J(U,X) + \frac{\partial}{\partial X}\mathcal{G}^T(U,X)\bar{U} \\
&= \frac{\partial}{\partial X}\mathcal{N}^T(U,\bar{U},X) \qquad \textbf{Mesh Adjoint equation} \\
\frac{d\mathcal{L}}{d\beta} &= \frac{d}{d\beta}\mathcal{M}^T(\beta)\bar{X} \qquad \textbf{Design equation}
\end{aligned}$$

Implementation

Application of AD in a mechanical fashion to the evaluation of objective function J directly yields gradients of the shifted Lagrangian $\mathcal{N}$:

$$\begin{aligned} U^{n+1} &= \mathcal{G}(U^n, X) \\ W &= J(U^n, X) \end{aligned}$$

Reverse Mode

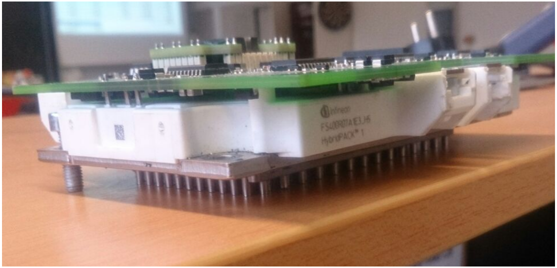
$$\begin{aligned} \bar{X} &= \partial J_X^T(U^n, X)\bar{W} \\ \bar{U}^n &= \partial J_U^T(U^n, X)\bar{W} \\ \bar{X} &\mathrel{+}= \partial \mathcal{G}_X^T(U^n, X)\bar{U}^{n+1} \\ \bar{U}^n &\mathrel{+}= \partial \mathcal{G}_U^T(U^n, X)\bar{U}^{n+1} \end{aligned}$$

If $\bar{W} \equiv 1$ and $U^n \equiv U^*$ we have

$$\begin{aligned} \bar{U}^{n+1} &\equiv \partial \mathcal{N}_U^T(U^*, \bar{U}^n, X), \\ \bar{X} &\equiv \partial \mathcal{N}_X^T(U^*, \bar{U}^n, X). \end{aligned}$$

Using the Expression Template approach we only need to store the gradient information of $\mathcal{G}$ and J once at $U^n = U^*$. Subsequent iterations only require a reverse sweep (Reverse Accumulation).

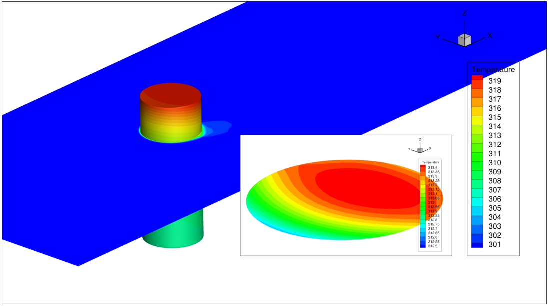
Conjugate Heat Transfer Applications



- Cooler consists of around 150 pins that extend into a coolant fluid flow
- Attached power electronic device (IGBT module, power loss around 600W due to internal resistance)
- All heat will be transferred through the pins – but at which temperature?

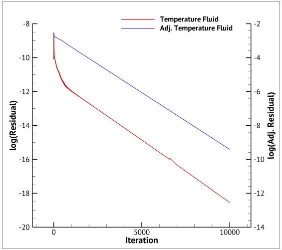
(Primal) Simulation – for one pin only

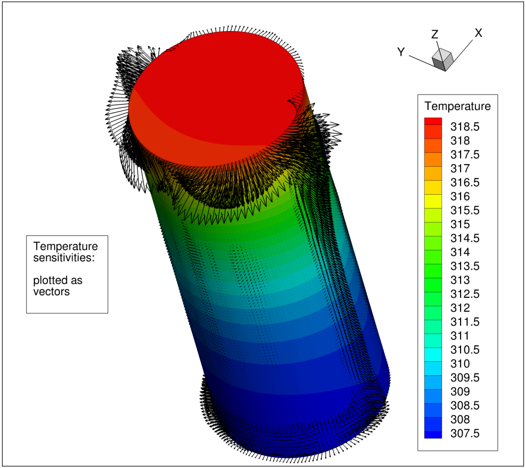
- (Steady) RANS fluid flow (water at $0.25 \frac{m}{s}$) with coupled heat equations in both fluid and solid zones in SU2
- Re $\sim$ 500, Prandtl-analogy for heat conduction, no viscous heating
- Heat flux at the pin's top: 4W, pin material: aluminium



Coupled Sensitivities

- **Objective function:**
  Temperature level at the top of the pins (minimize to avoid damage to power electronics!)
- **Adjoints:**
  Capture the coupling (and turbulence!) dependence
- **Sensitivities:**
  Include the mesh deformation derivatives

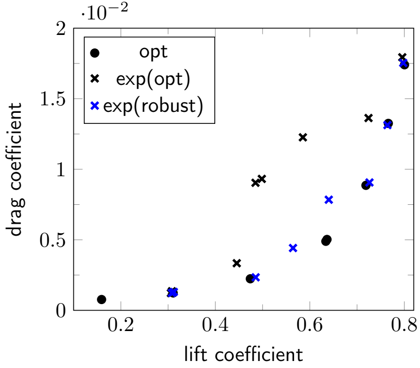


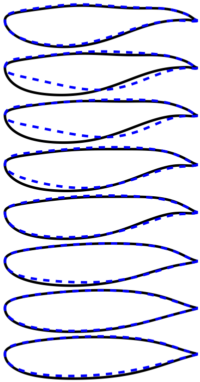


Robust Design with Multiple Objectives

Application in SU2

- uncertainty in the airfoil geometry given by a random field: non-intrusive pseudo-spectral approach + dimension-adaptive sparse grid
- steady Euler optimization test case

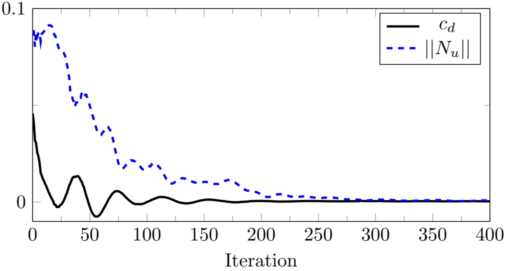


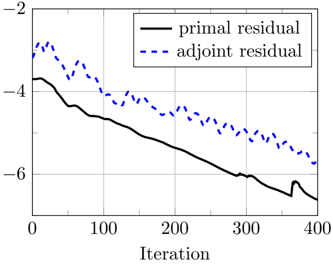


One-Shot Approach

Implementation in SU2

- simultaneous iteration of state, adjoint state and design
- research on additional constraints, topology optimization





- drag coefficient (transonic Euler flow)
- end compliance of cantilever beam (nonlinear continuum mechanics)

Summary

Discrete Adjoint Solver using AD

- **Easily extensible** to other (coupled) solvers in SU2 (more examples are shown in some of the next talks)
- **Fully parallel and future-proof** implementation
- **High-performance** (typically runtime and memory ratios of 1.0 - 2.0 and 4-6, respectively)

Thank you for your attention!
Any questions ?