A Unifying Computational Framework for Adaptive High-Order Finite Element Methods

Michael Woopen, Aravind Balan, and Georg May
Graduate School AICES, RWTH Aachen University

June 23rd, 2015
Motivation

- Methods and applications receive most of the attention in talks and publications
- The link between these, the implementation, is often neglected

Topics

- General structure of our computational framework
- Modularism via C++ templates and object orientation
- Algorithmic differentiation for speed and implementational ease
AdHoCFD (Adaptive High-Order CFD)

- (Primal & Mixed) HDG & DG
- Newton-Krylov with pseudo-time-stepping, line search, and physical constraints
- GMRES with ILU(n) via PETSc
- Written in C++ using templates
- AD for analytical Jacobian
- Euler, Navier-Stokes, RANS ($k$-$\omega$ & S-A) in 2d & 3d
- Mixed element meshes
- Isotropic & anisotropic $hp$-adaptation (based on the adjoint)
Structure (1/3)

AdHoCFD

Physical Model

Adaptation

Discretization

Linear Solver

Nonlinear Solver
### Physical Model
- Fluxes, etc.
- Boundary and initial conditions
- Outputs

### Discretization
- Basis functions
- Assembly
- Numerical fluxes
Structure (3/3)

Linear Solver
- Vector & matrix operations
- Iterative solvers
- Preconditioner

Nonlinear Solver
- Continuation strategies
- Line-searches
- Constraints
C++ Templates

- Generic implementation of functions and classes
- Specialization at compile-time
- Enables major optimizations via the compiler

Example

```cpp
template <typename Type>
Type max(Type a, Type b) {
    return a > b ? a : b;
}
```
C++ Templates: Discretizations

template <int D, int C, class Model, class LinearSolver>
class HDG: public Discretization<D, C, Model, LinearSolver> {
    public:
    void Preprocessing();

    void AssembleSystem(Solution & sol);
    void AssembleResidual(Solution & sol);

    void SolveLinearizedSystem(Solution & sol, Solution & delta);
    void UpdateSolution(Solution & sol, Solution & delta);

    void LoadParameters();

    ...
};
template <int D>
class CompressibleEuler : public DummyModel<D+2, D> {

public:

    // Characterization of the model
    const bool Convection = true;
    const bool Diffusion  = false;
    const bool Source     = false;

    void EvalConvFlux(Vec<D+2> & state, Vec<D> & pos, Mat<D+2, D> & res);

    void EvalBdryState(int bc, Vec<D+2> & state, Vec<D> & pos, Vec<D> & normal, Vec<D+2> & res);

    void LoadParameters();

    ...
};
Algorithmic Differentiation (AD)

**Jacobians**

- Necessary in implicit methods
- Obtained via
  - Manual differentiation: (potentially) fast, exact, tedious, error-prone
  - Finite differences: easy, inaccurate (cancelation)
  - Algorithmic differentiation: fast, easy, exact

**How does it work?**

- Create new data type which carries both its value and its derivative
- Overload operators (+, -, *, /, ...) for that data type
- Derivatives are automatically computed in the background
template <int N, typename SCAL = double>
class AD {

    SCAL val;
    SCAL dval[N];

public:
    // returns value
    SCAL Value() const { return val; }

    // returns partial derivative
    SCAL DValue (int i) const { return dval[i]; }

    // assign constant value
    AD & operator= (SCAL aval);

    ...
};
template<int N, typename SCAL> 
AD<N, SCAL> operator+ (const AD<N, SCAL> & x, const AD<N, SCAL> & y) {

    AD<N, SCAL> res;
    res.Value () = x.Value() + y.Value();

    for (int i = 0; i < N; ++i)
        res.DValue(i) = x.DValue(i) + y.DValue(i);

    return res;
}
template<int N, typename SCAL>
AD<N, SCAL> operator* (const AD<N, SCAL> & x, const AD<N, SCAL> & y) {

    AD<N, SCAL> res;
    SCAL hx = x.Value();
    SCAL hy = y.Value();

    res.Value() = hx * hy;

    for (int i = 0; i < N; ++i)
        res.DValue(i) = hx * y.DValue(i) + hy * x.DValue(i);

    return res;
}
template <int D>
void CompressibleEuler<D>
::EvalConvFlux(Vec<D+2> & state, Vec<D> & pos, Mat<D+2, D> & res) {

    double rho = state(0);
    Vec<D> m = state.Rows(1, D+1);
    double rhoE = state(D+1);
    Vec<D> U = 1. / rho * m;

    double U2 = InnerProduct(U, U);
    double p = (gamma - 1.) * (rhoE - 0.5 * rho * U2);

    res = state * Trans(U);
    res.Rows(1, D+1) += p * Id<D>();
    res.Row(D+1) += p * U;
}

template <int D>
template <typename SCAL>
void CompressibleEuler<D>::EvalConvFlux(Vec<D+2, SCAL> & state, Vec<D> & pos,
                                                  Mat<D+2, D, SCAL> & res) {

    SCAL rho = state(0);
    Vec<D, SCAL> m = state.Rows(1, D+1);
    SCAL rhoE = state(D+1);
    Vec<D, SCAL> U = 1. / rho * m;

    SCAL U2 = InnerProduct(U, U);
    SCAL p = (gamma - 1.) * (rhoE - 0.5 * rho * U2);

    res = state * Trans(U);
    res.Rows(1, D+1) += p * Id<D>();
    res.Row(D+1) += p * U;
}
Bringing it together

Build separate solvers, i.e.

\[
\text{AcHoCFD<HDG, CompressibleEuler, Newton, Petsc, 3> hdg\_euler3;}
\]

or

\[
\text{AcHoCFD<DG, CompressibleSA, Newton, Petsc, 2> dg\_sa2;}
\]

Build several solvers into one executable, i.e.

\[
\begin{align*}
\text{AcHoCFD<HDG, CompressibleEuler, Newton, Petsc, 2> hdg\_euler2;} \\
\text{AcHoCFD<HDG, CompressibleNavierStokes, Newton, Petsc, 2> hdg\_ns2;} \\
\text{AcHoCFD<HDG, CompressibleSA, Newton, Petsc, 2> hdg\_sa2;} \\
\text{AcHoCFD<HDG, CompressibleKOmega, Newton, Petsc, 2> hdg\_kw2;} \\
\text{AcHoCFD<DG, CompressibleEuler, Newton, Petsc, 2> dg\_euler2;} \\
\text{AcHoCFD<DG, CompressibleNavierStokes, Newton, Petsc, 2> dg\_ns2;} \\
\text{AcHoCFD<DG, CompressibleSA, Newton, Petsc, 2> dg\_sa2;} \\
\text{AcHoCFD<DG, CompressibleKOmega, Newton, Petsc, 2> dg\_kw2;}
\end{align*}
\]
### Numerical Results

#### Cases

- Turbulent flat plate (NASA)
- Turbulent backward facing step (NASA)
- Turbulent RAE 2822 (HIOCFD)
- Laminar delta wing (HIOCFD)
Turbulent Flat Plate

Setup

- $Ma = 0.2$, $Re_1 = 5 \cdot 10^6$, S-A turbulence model
- Structured meshes with 120, 480, 1920, and 7680 quad elements
- Adaptive results starting from an isotropic 1106 element mesh
Turbulent Flat Plate

Setup

- $Ma = 0.2$, $Re_1 = 5 \cdot 10^6$, S-A turbulence model
- Structured meshes with 120, 480, 1920, and 7680 quad elements
- Adaptive results starting from an isotropic 1106 element mesh
Turbulent Flat Plate

Setup

- $\text{Ma} = 0.2$, $\text{Re}_1 = 5 \cdot 10^6$, S-A turbulence model
- Structured meshes with 120, 480, 1920, and 7680 quad elements
- Adaptive results starting from an isotropic 1106 element mesh
Turbulent Flat Plate: Skin Friction

\[ n_e = 120 \]
Turbulent Flat Plate: Skin Friction

\[ n_e = 480 \]
Turbulent Flat Plate: Skin Friction

\[ n_e = 1920 \]

\[ p = 1 \]
\[ p = 2 \]
\[ p = 3 \]
\[ p = 4 \]
Turbulent Flat Plate: Skin Friction

\[ n_e = 7680 \]
Turbulent Flat Plate: Skin Friction

Adapted result

$\frac{p=3, \ n_e = 4397}{p=4, \ n_e = 7680}$
Turbulent Flat Plate: Adapted Meshes
Turbulent Backward Facing Step

Setup

- $Ma = 0.128$, $Re_1 = 36000$, S-A turbulence model
- Structured meshes with 4992 and 19968 quad elements
- Adaptive results starting from an isotropic 1920 element mesh
Turbulent Backward Facing Step

Setup

- $Ma = 0.128$, $Re_1 = 36,000$, S-A turbulence model
- Structured meshes with 4,992 and 19,968 quad elements
- Adaptive results starting from an isotropic 1920 element mesh
Turbulent Backward Facing Step

Setup

- \( Ma = 0.128, \, Re_1 = 36 000 \), S-A turbulence model
- Structured meshes with 4,992 and 19,968 quad elements
- Adaptive results starting from an isotropic 1920 element mesh
Turbulent Backward Facing Step: Skin Friction

\[ n_e = 4992 \]
Turbulent Backward Facing Step: Skin Friction

\[ n_e = 19968 \]
Turbulent Backward Facing Step: Skin Friction

Adaptive Result

\[ c_f \]

\[ p=3, n_e = 4396 \]

\[ p=4, n_e = 19968 \]

Adaptive Result
Turbulent Backward Facing Step: Mach Number
Turbulent Backward Facing Step: Adapted Meshes

\( p = 1 \)
Turbulent Backward Facing Step: Adapted Meshes

$p = 3$
Turbulent RAE 2822

Setup

- $Ma = 0.734$, $Re_c = 6.5 \cdot 10^6$, $\alpha = 2.79^\circ$, S-A turbulence model
- Meshes with 4048, 16192, and 64768 triangles
Turbulent RAE 2822: Mach Number

\[ n_e = 4048, \ p = 1 \]
\[ \Rightarrow n_{\text{dof}} = 12\ 144 \]

\[ n_e = 16192, \ p = 3 \]
\[ \Rightarrow n_{\text{dof}} = 40\ 480 \]
Turbulent RAE 2822: Mach Number

\[ n_e = 64768, \ p = 1 \]
\[ \Rightarrow n_{\text{dof}} = 194\ 304 \]

\[ n_e = 16192, \ p = 3 \]
\[ \Rightarrow n_{\text{dof}} = 161\ 920 \]
Laminar Delta Wing

Setup

- \( Ma = 0.3, \ Re_c = 4000, \ \alpha = 12.5^\circ \)
- Meshes with 408, 3264, and 26112 hexahedrons
Laminar Delta Wing — Drag

\[ |\Delta C_d| \]

\[ \begin{align*}
p = 1 & \quad \text{(blue dot)} \\
p = 2 & \quad \text{(red square)} \\
p = 3 & \quad \text{(brown dot)} \\
p = 4 & \quad \text{(black asterisk)}
\end{align*} \]

\[ \frac{1}{\text{ndof}^{1/3}} \]

July 22nd, 2014  Woopen, Balan, May  Page 28/30  A Unifying Computational Framework for FE Methods
Laminar Delta Wing — Lift

\[ |\Delta C_l| \]

\[ 10^{-1} \]

\[ 10^{-2} \]

\[ 10^{-3} \]

\[ \frac{1}{\text{ndof}^{1/3}} \]

\[ 10^{-1} \]

$p = 1$

$p = 2$

$p = 3$

$p = 4$
Conclusion

- Structural overview of our computational framework
- C++ templates for modularism
- Algorithmic differentiation for speed and implementational ease
- Promising results for various test cases