We present a comprehensive overview of our computational framework for adaptive high-order finite element methods, including discontinuous Galerkin (DG) methods and their hybridized counterparts (HDG). Besides covering the numerical methods, we grant their actual implementation a prominent position in this paper. Finally, we apply our framework to a variety of flow problems, including laminar, and turbulent flow in both two- and three-dimensional domains.

I. Introduction

In a series of papers, we have presented the extension of our computational framework towards more complex physics and more efficient methods, resulting in a unifying computational framework including both DG and HDG methods, and various adaptation techniques. This paper shall address first and foremost details on the structure and implementation of our framework. Employing techniques from both generic and object-oriented programming, the framework is designed with efficiency and extensibility in mind. Typical components like physical models, discretizations, or linear solvers can be exchanged rapidly. In order to reduce implementational complexity even further, algorithmic differentiation is used to obtain derivatives of fluxes and boundary conditions necessary for an implicit discretization.

Often, specific parts of a computational framework can be separated from each other into distinct modules. Intermodular communication occurs via specified interfaces which are kept as simple and general as possible. This way, one component, e.g. a discretization, can be swapped with an alternative implementation very easily as long as compatibility with respect to the interfaces is ensured. Thus, code redundancy is decreased and ease of adding new components is increased. Both these factors contribute to a less error-prone and more efficient implementational environment.

Algorithmic differentiation (AD) is an attractive way to obtain derivatives of functions. Although the alternative, implementing these derivatives manually, might reduce the overall runtime, its implementational complexity is substantial for non-trivial functions. AD can be realized in different ways. We make use of a technique called operator overloading. Here, a special data type, carrying not only its value but also its derivative with respect to certain inputs, is defined. Additionally, necessary algebraic operations on this data type and their implications on derivatives are specified. Functions to be differentiated are then implemented with a generic data type for input and output arguments. This way the compiler can generate various realizations of one function, e.g. using conventional real-valued or a AD data types. Hence, the developer only specifies the function itself, never its derivative.

The structure of this paper is given in the following. We begin by giving a condensed description of the numerical methods being used. Before we explain the components of our framework in particular, we illustrate its overall structure and the interplay between components. Having the big picture in mind, we turn to a detailed description of physical models and discretizations, and their efficient implementation. Finally, we apply our framework to test cases including laminar and turbulent flow.
II. Methods

We consider systems of balance laws

\[ \nabla \cdot (f_c(w) - f_v(w, \nabla w)) = s(w, \nabla w) \]  (1)

with given convective and diffusive fluxes

\[ f_c : \mathbb{R}^m \to \mathbb{R}^{m \times d}, \quad f_v : \mathbb{R}^m \times \mathbb{R}^{m \times d} \to \mathbb{R}^{m \times d}, \]  (2)

and a state-dependent source term

\[ s : \mathbb{R}^m \times \mathbb{R}^{m \times d} \to \mathbb{R}^m. \]  (3)

We denote the spatial dimension by \( d \) and the number of conservative variables by \( m \). For \( f_v \neq 0 \) or a gradient-dependent source term \( s \), we may formally rewrite (1) as

\[ \mathbf{q} = \nabla w \]

\[ \nabla \cdot (f_c(w) - f_v(w, \mathbf{q})) = s(w, \mathbf{q}). \]  (5)

Formulation (4)–(5) is frequently applied when motivating viscous discontinuous Galerkin discretizations, as it constitutes a system of only first-order partial differential equations.12,13

A. Notation

We tessellate the domain \( \Omega \) into a collection of non-overlapping elements, denoted by \( T_h \), such that \( \bigcup_{K \in T_h} K = \Omega \). For the element edges we consider two different kinds of sets, \( \partial T_h \) and \( \Gamma_h \), which are element-oriented and edge- or face-oriented, respectively:

\[ \partial T_h := \{ \partial K \setminus \partial \Omega : K \in T_h \}, \]

\[ \Gamma_h := \{ e : e = K \cap K' \text{ for } K, K' \in T_h; \text{meas}_{d-1}(e) \neq 0 \}. \]  (7)

The first is the collection of all element boundaries, which means that every edge or face appears twice. The latter, however, includes every element interface just once. The reason for this distinction will become clear later. Please note that neither of these sets shall include edges or faces lying on the domain boundary; the set of boundary edges is denoted by \( \Gamma_b^h \).

We denote by \( \Pi^p(D) \) the set of polynomials of degree at most \( p \) on some domain \( D \). We will need discontinuous function spaces for the domain and the mesh skeleton:

\[ V_h = \{ v \in L^2(\Omega) : v|_K \in \Pi^K(K), \ K \in T_h \}^{m \times d} \]  (8)

\[ W_h = \{ w \in L^2(\Omega) : w|_K \in \Pi^K(K), \ K \in T_h \}^m \]  (9)

\[ M_h = \{ \mu \in L^2(\Gamma_h) : \mu|_e \in \Pi^K(e), \ e \in \Gamma_h \}^m. \]  (10)

Thus, \( v \in V_h, w \in W_h \) and \( \mu \in M_h \) are piecewise polynomials of degree \( p \) which can be discontinuous across edges (for \( v, w \)) or vertices (for \( \mu \)), respectively.

We will distinguish between element-oriented inner products (defined with \( T_h \)), i.e.

\[ (v, w)_{T_h} := \sum_{K \in T_h} \int_K vw \, dx, \]

\[ (v, w)_{T_h} := \sum_{K \in T_h} \int_K v \cdot w \, dx, \]

\[ (v, w)_{\partial T_h} := \sum_{K \in T_h} \int_{\partial K} vw \, d\sigma, \]  (11)

and edge- or face-oriented inner products (defined with \( \Gamma_h \)), i.e.

\[ \langle v, w \rangle_{\Gamma_h} := \sum_{e \in \Gamma_h} \int_e vw \, d\sigma. \]  (12)
B. HDG discretization

Find \( \mathbf{x}_h := (\mathbf{q}_h, w_h, \lambda_h) \in \mathcal{X}_h := (\mathbf{V}_h, W_h, M_h) \) s.t. \( \forall \mathbf{y}_h := (\mathbf{r}_h, \varphi_h, \mu_h) \in \mathcal{X}_h \)

\[
0 = \mathcal{N}_h (\mathbf{q}_h, w_h, \lambda_h; \mathbf{r}_h, \varphi_h, \mu_h) \\
:= (\mathbf{r}_h, \mathbf{q}_h)_{\mathcal{T}_h} + (\nabla \cdot \mathbf{r}_h, w_h)_{\mathcal{T}_h} - (\mathbf{r}_h \cdot \mathbf{n}, \lambda_h)_{\partial \mathcal{T}_h} \\
- ((\nabla \varphi_h, \mathbf{f}_c (w_h) - \mathbf{f}_c (w_h, \mathbf{q}_h)) - (\varphi_h, s (w_h, \mathbf{q}_h))_{\mathcal{T}_h} + \langle \varphi_h, \hat{\mathbf{f}} - \mathbf{f}_c \rangle_{\partial \mathcal{T}_h} \\
+ \mathcal{N}_{h, \partial \Omega} (\mathbf{q}_h, w_h; \varphi_h) + \mathcal{N}_{h, \partial \Omega} (\mathbf{q}_h, w_h; \mathbf{r}_h, \varphi_h) \\
+ \langle \mu_h, \left[ \hat{\mathbf{f}} - \mathbf{f}_c \right] \rangle_{\mathcal{T}_h}.
\]

In order to realize a consistent and stable method, a suitable choice of the fluxes is as important as the appropriate choice of the shock-capturing operator \( \mathcal{N}_{h, \partial \Omega} (\mathbf{q}_h, w_h; \varphi_h) \) and the boundary operator \( \mathcal{N}_{h, \partial \Omega} (\mathbf{q}_h, w_h; \mathbf{r}_h, \varphi_h) \).

The convective flux is stabilized with a Roe-like approach,\(^1\) i.e.

\[
\hat{\mathbf{f}}_c (\lambda_h, w_h) = \mathbf{f}_c (\lambda_h) \cdot \mathbf{n} - |\mathbf{f}'_c (\lambda_h)| (\lambda_h - w_h)
\]

where the eigen-decomposition of the convective flux Jacobian evaluated with \( \lambda_h \) is used, i.e.

\[
|\mathbf{f}'_c (\lambda_h) \cdot \mathbf{n}| = R (\lambda_h, \mathbf{n}) |\Lambda (\lambda_h, \mathbf{n})| R^{-1} (\lambda_h, \mathbf{n}.
\]

For the diffusive flux we use a BR2-like method,\(^2\) i.e.

\[
\hat{\mathbf{f}}_c (\lambda_h, w_h, \mathbf{q}_h) = (\mathbf{f}_c (\lambda_h, \mathbf{q}_h) + \mathbf{r} (\mathbf{K} (\lambda_h) (\lambda_h - w_h) \mathbf{n})) \cdot \mathbf{n}.
\]

where we assume that the viscous flux is linear in the gradient, i.e. can be written as

\[
\mathbf{f}_c (w, \mathbf{q}) = \mathbf{K} (w) \mathbf{q}, \quad \mathbf{K} \in \mathbb{R}^{m \times d \times m \times d}.
\]

The lifting operator \( \mathbf{r} \) on an element \( K \) with respect to an edge \( e \) is then defined as

\[
(\mathbf{r}_h, \mathbf{K} (\lambda_h) (\lambda_h - w_h) \mathbf{n})_{\mathcal{K}} = \eta_e (\mathbf{r}_h, \mathbf{K} (\lambda_h) (\lambda_h - w_h) \mathbf{n})_e.
\]

In order to retrieve an adjoint-consistent scheme, special care has to be taken when discretizing the boundary conditions;\(^3\) they have to be incorporated by using the boundary states \( w_{\partial \Omega} (w_h) \) and gradients \( q_{h, \partial \Omega} (w_h, \mathbf{q}_h) \), i.e.

\[
\mathcal{N}_{h, \partial \Omega} (\mathbf{q}_h, w_h; \mathbf{r}_h, \varphi_h) := (\mathbf{r}_h \cdot \mathbf{n}, w_{\partial \Omega} (w_h))_{\mathcal{T}^i_h} \\
+ \langle \varphi_h, (\mathbf{f}_c (w_{\partial \Omega}) - \mathbf{f}_c (w_{\partial \Omega}, \mathbf{q}_h, \partial \Omega) - \mathbf{r} (\mathbf{K} (w_{\partial \Omega}) (w_{\partial \Omega} - w_h)) \cdot \mathbf{n} \rangle_{\mathcal{T}^i_h}.
\]

Please note, that viscous stabilization is also present at boundaries. As long as boundary functionals are evaluated accordingly, the method remains adjoint-consistent. Furthermore, we would like to emphasize that \( \lambda_h \) does not occur in this boundary term as it is only defined on interior faces.

For more details on the hybridization of the system, we refer to our earlier papers.\(^1,2,3,4,5,6,7,8\)

C. Adjoint-based error estimation

In engineering applications, usually only a few scalar quantities are of interest, and not necessarily the solution quality per se. In aerodynamic applications, these quantities can, e.g., be lift and drag.

Adjoint-based error estimation has been developed in order to most accurately approximate such quantities.\(^9,10\) To put this into a mathematical context, let a functional \( \mathcal{J}_h : \mathcal{X}_h \rightarrow \mathbb{R} \) be given and consider the error in this quantity,

\[
e_h := \mathcal{J}_h (\mathbf{x}) - \mathcal{J}_h (\mathbf{z}_h).
\]

For the derivation of the adjoint-based error estimate we expand the target functional in a Taylor series as follows

\[
\mathcal{J}_h (\mathbf{x}) - \mathcal{J}_h (\mathbf{z}_h) = \mathcal{J}'_h [\mathbf{z}_h] \delta \mathbf{x}_h + \mathcal{O} (||\delta \mathbf{x}_h||^2)
\]
where we defined $\delta x_h := x - x_h$.

We proceed in a similar manner with the error in the residual, i.e.

$$ N_h (x ; y_h) - N_h (x_h ; y_h) = N_h' [x_h] (\delta x_h ; y_h) + O (\| \delta x_h \|^2). \quad (20) $$

As our discretization is consistent the first term $N_h (x ; y_h)$ vanishes.

Substituting Eq. (20) into Eq. (19) and neglecting the quadratic terms yields

$$ e_h \approx \eta := - N_h (x_h ; z_h) \quad (21) $$

where $z_h$ is defined by the so-called adjoint equation

$$ N_h' [x_h] (y_h ; z_h) = J_h' [x_h] (y_h) \quad \forall y_h \in \tilde{X}_h. \quad (22) $$

The adjoint solution $z_h = \left( q_h, \bar{w}_h, \bar{\lambda}_h \right) \in \tilde{X}_h$ represents the link between variations in the residual and in the target functional.

The global error estimate $\eta$ can then be restricted to a single element to yield a local indicator to drive an adaptation procedure, i.e.

$$ \eta_K := \left| N_h (x_h ; z_h) \right|_K. \quad (23) $$

Please note, that the functionals $N_h$ and $J_h$ and their jacobians have to be evaluated in a somewhat richer space than $X_h$, namely $\tilde{X}_h \supset X_h$. Otherwise the weighted residual $N_h (x_h ; z_h)$ would be identical zero as

$$ N_h (x_h ; y_h) = 0 \quad \forall y_h \in X_h. \quad (24) $$

This can be achieved by either mesh refinement or a higher polynomial degree of the ansatz functions. In our setting, especially when using a hierarchical basis, the latter is advantageous with respect to implementational effort and efficiency.

D. Mesh adaptation

Our framework includes both isotropic\,\textsuperscript{4,5} and anisotropic mesh-refinement. The latter computes anisotropic information (principal directions, stretching) in order to minimize the local interpolation error which is approximated by a reconstructed higher-order derivative. The desired isotropic element size is then found via the adjoint-based error estimate using a fixed-fraction approach.\textsuperscript{22} Finally, the new mesh is generated with the Bidimensional Anisotropic Mesh Generator (BAMG).\textsuperscript{23} An extension to anisotropic $hp$-adaptation can be found in a concurrent paper.\textsuperscript{24}

III. Implementation

Our computational framework is built in a modular fashion where each module is represented as a class and has to implement certain interfaces. This way, single modules can be exchanged very easily. The major modules can be seen in Fig. 1. Their structure and functionality will be explained in the following sections.

A. Algorithmic differentiation

Before we describe the separate modules of our framework, we want to give a small introduction into algorithmic differentiation using operator overloading. In our framework, we only apply AD to functions where the number of inputs and outputs is known at compile time (e.g. fluxes, boundary conditions, etc.). Hence, the AD class (see Lst. 1), which carries not only a value but also its derivatives can be templated with the number of derivatives as a parameter.

```cpp
template <int N, typename SCAL = double>
class AD {
  SCAL val;
  SCAL dval[N];
public:
  // returns value
  SCAL Value() const { return val; }
};
```

\textsuperscript{1} 4 of 22

American Institute of Aeronautics and Astronautics
In order for AD-type variables to be used in a standard way, mathematical operations have to be overloaded for this kind of variable. In Lst. 2, we show functions for the addition and multiplication of AD-type variables. The implementation of these functions is straightforward: one only has to apply differentiation rules to the mathematical operation at hand (e.g. sum rule, product rule, etc.). As a result, the differentiation of functions is carried out automatically in the background, not visible for the developer.
B. Physical models

A model in our setting is defined by its fluxes (and possibly a source function), initial and boundary conditions. In addition to these, a model can have certain monitors, for example weighted boundary fluxes like lift and drag in flow problems. In Lst. 3, an excerpt from the model describing the compressible Euler equations is given. There are several observations to be made. The whole class is templated with a parameter $D$ which represents the spatial dimension and can be used by every function inside the class. If a member function of CompressibleEuler<2> is called, the compiler automatically generates the necessary functions with $D = 2$. This renders the implementation of functions for every space dimension redundant. Furthermore, we make heavy use of matrix and vector classes (intended for small dimensions). Thereby, we can implement the Euler flux function compactly yet visually appealing (see Lst. 4). The most important fact, however, has not been mentioned yet. Functions representing fluxes or boundary conditions are templated themselves with a type SCAL used for input and outputs. The simplest type to think of is a floating point number. In that case, the function would return just the expected function value. If one uses, on the other hand, an AD data type, the function returns the function value and its derivative with respect to its inputs. Hence, there is no need to implement a separate function for derivatives.

In addition to that, a model has to contain information on which kind of fluxes it has. The compressible Euler equations have only a convective component. Based on that the compiler decides which parts of the discretization to build. Furthermore, it can contain user-defined parameters used for boundary conditions or material definitions.

Our framework comprises the compressible Euler, Navier-Stokes and Reynolds-averaged Navier-Stokes equations. The latter can be combined with the $k$-$\omega$ and the negative Spalart-Allmaras turbulence models. The implementation of these models are built on top of each other, i.e. the Navier-Stokes class inherits from the Euler class, and the RANS classes inherit from the Navier-Stokes class (see Lst. 5). By doing so, code redundancy can be decreased quite a lot.
template <int D>
class CompressibleEuler {
 public:

 // Characterization of the model
 static const int Components = D+2;
 static const int Dimension = D;
 static const bool Convection = true;
 static const bool Diffusion = false;
 static const bool Source = false;

 // User-defined parameters
 static double mach;
 static Vec<D-1> angle;
 static double gamma;
 static double chord;

template <typename SCAL>
static void 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.Row(0) = m;
  res.Rows(1,D+1) = m * Trans(U) + p * Id<D>();
  res.Row(D+1) = (rhoE + p) * U;
}

Listing 3: Model class for the compressible Euler equations

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.Row(0) = m;
  res.Rows(1,D+1) = m * Trans(U) + p * Id<D>();
  res.Row(D+1) = (rhoE + p) * U;
}

Listing 4: Implementation of Euler flux
C. Discretization

As a basis for every discretization, we use the Netgen/Ngsolve library\textsuperscript{26} which offers geometry handling and mesh generation, and quadrature rules and the evaluation of basis functions for numerous finite elements. Thus, the discretization classes themselves are only responsible for the assembly of both the primal and (potentially) dual system (see Lst. 6). If an implicit relaxation technique or time integration is used, it can be equipped with a linear solver class (see Sec. E and Lst. 7). For an explicit time integration, the discretization offers a routine to compute solely the residual.

```cpp
template <class Model, class LinearSolver>
class Discretization {
public:
    static void Preprocessing();
    static void AssembleSystem(Solution & sol);
    static void AssembleResidual(Solution & sol);
    static void SolveLinearizedSystem(Solution & sol, Solution & delta,
        double res_min, int & nit, double & res);
    static void SolveAdjointSystem(Solution & sol, Solution & adj, int & nit, double & res);
    static void UpdateSolution(Solution & sol, Solution & delta);
    static void LoadParameters();
};
```

Listing 6: Exemplary discretization class

```cpp
template <class Model, class LinearSolver>
void Discretization<Model, LinearSolver>::SolveLinearizedSystem(Solution & sol, Solution & delta,
    double res_min, int & nit, double & res) {
    LinearSolver::ResetMatrix();
    LinearSolver::ResetRhs();
    // Assemble matrix and rhs
    AssembleSystem(sol);
    // Check for convergence and solve the system
    LinearSolver::GetResidualNorm(res);
    if (res > res_min)
        LinearSolver::Solve(delta, nit);
}
```

Listing 7: Solution of linearized system
D. Nonlinear solver

The nonlinear solver class (see Lst. 8) serves as the main driver. The entry point is the \texttt{Solve} function. Usually, we employ a Newton solver (see Lst. 9), which calls the discretization to assemble the linear system, solves it, and updates the solution until a certain convergence criterion is met. In order to enhance robustness, constraints and line-searches can be used.

```cpp
template <class Model, class Discretization>
class Newton {
public:
  static void Solve(Solution & sol);
  static void CheckConstraints(Solution & sol, Solution & delta);
  static void LineSearch(Solution & sol, Solution & delta);
  static void WriteOutput(Solution & sol, int nit_lin, double res);
  static void LoadParameters();
};
```

Listing 8: Newton solver class

```cpp
template <class Model, class Discretization>
void Newton<Model, Discretization>::Solve(Solution & sol) {
  Solution delta;
  int nit_lin;
  double res;
  for (int i = 0; i < nit_nonlin && res > res_min; ++i) {
    Discretization::SolveLinearizedSystem(sol, delta, res_min, nit_lin, res);
    // Physical constraints and line-search for robustness
    if (checkconstraints) CheckConstraints(sol, delta);
    if (linesearch) LineSearch(sol, delta);
    Discretization::UpdateSolution(sol, delta);
  }
  WriteOutput(sol, nit_lin, res);
}
```

Listing 9: Newton driver

E. Linear solver

The linear solver class represents a very high-level interface to either an external or in-house linear algebra library. It provides access to matrix and vector assembly routines as well as linear solvers and preconditioners (see Lst. 10). Hence, it is only visible to the discretization class. We use PETSc\textsuperscript{27,28} in our work.
class LinearSolver {
public:
static void Init();
static void ResetMatrix();
static void ResetRhs();
static void AddMatrix(int i, int j, Matrix & mat);
static void AddRhs(int i, Vector & vec);
static void Solve(Solution & delta, int & nit, double & res, bool transpose = false);
static void LoadParameters();
};

Listing 10: Linear solver class

IV. Results

A. Turbulent boundary layer on a flat plate

In this test case, we consider subsonic, turbulent flow along a flat plate. We employ the Spalart-Allmaras turbulence model. The free stream Mach number is \( M_a = 0.2 \), and the Reynolds number based on the length of the plate is \( Re_L = 10^7 \). The angle of attack is \( \alpha = 0^\circ \). We prescribe the SA working variable in the free-stream as \( \nu / \nu_\infty = 3.29 \).

An adiabatic plate of length 2 is placed between symmetry planes of the same length. The top boundary is located at \( y = 4 \). We prescribe Riemann invariant in- and outflow boundary conditions on the left and right boundary, respectively. The top boundary is modeled as a symmetry plane.

We perform both uniform and adaptive refinement studies. The structured mesh family ranges from 120 to 7 680 quadrilateral elements (see Fig. 2). Theses meshes are refined towards the wall and both leading and trailing edge. For the adaptive computations we use drag as the target functional of interest. The initial mesh for adaptive refinement can be seen in Fig. 3. It consists of 1 106 triangular elements and provides no boundary layer resolution. It is only mildly refined towards the wall.

![Figure 2: Coarsest and finest flat plate mesh.](image)

Skin friction is a very good indicator for mesh resolution as it heavily relies on both near-wall refinement and element clustering at leading and trailing edge. In Fig. 4, the skin friction coefficient is given for \( p = 1 \ldots 4 \) on the meshes provided by NASA. Besides these meshes, they also provide correlations for skin
friction towards leading and trailing edge:

\[ c_{f,le} = 2.503 \cdot 10^{-4} \cdot x^{-0.25}, \]  
\[ c_{f,te} = 2.7001033 \cdot 10^{-3} \cdot x^{-0.145} + 7 \cdot 10^{-6} \cdot (2 - x)^{-0.5}. \]  

They are plotted with dashed lines next to our results. One can see that both mesh refinement and order increase result in a better skin friction approximation. Oscillations are present at leading and trailing edge, but restricted to the first (or last) element. The overall behavior of skin friction along the plate is very smooth.

In Fig. 5, three meshes from a \( p = 3 \) adaptation can be seen. They all consist of large elements further away from the plate and very thin elements along the plate. By taking a closer look at the near-wall behavior of these meshes, one can depict four important features. The most obvious one, which has already been mentioned is the grading towards the wall. Furthermore, the adaptive process clusters element at both leading and trailing edge of the plate. Last, a feature not so obvious, is refinement along the boundary layer edge. The SA working variable exhibits a strong gradient in that region which has to be resolved in order to prevent oscillations. In Fig. 6, we show skin friction computed on these meshes. With increasing elements, the singularities at leading and trailing edge are increasingly resolved.
Figure 4: Skin friction coefficient with focus on both leading and trailing edge of the turbulent flat plate.
Figure 5: Adapted flat plate meshes ($p = 3$).

(a) $n_e = 397$

(b) $n_e = 397$, vertical zoom

(c) $n_e = 1516$

(d) $n_e = 1516$, vertical zoom

(e) $n_e = 6471$

(f) $n_e = 6471$, vertical zoom
B. Turbulent flow over a backward facing step

Here, we consider subsonic, turbulent flow over a backward facing step. We employ the Spalart-Allmaras turbulence model. The free stream Mach number is $Ma = 0.128$, and the Reynolds number based on the step height is $Re_H = 36,000$. We prescribe the SA working variable in the free-stream as $\hat{\nu}_\infty / \nu_\infty = 3$.29

In this case, a turbulent boundary layer encounters a sudden back step, causing flow separation. The flow then reattaches and recovers downstream of the step. The inflow is located at $x = -130$, followed by a symmetry plane, and finally the wall at $x = -110$. The step itself is located at $x = 0$ and has unit height. The outflow is placed at $x = 50$. The channel has height 8 before and 9 after the step. We prescribe Riemann invariant in- and outflow boundary conditions on the left and right boundary, respectively.

We perform both uniform and adaptive refinement studies. The structured meshes have 4 992 and 19 968 quadrilateral elements (see Fig. 7). These meshes provide a very good boundary layer resolution towards horizontal walls. The clustering towards the step is not as strong. For the adaptive computations we use drag along the wall downstream of the step as the target functional of interest. The initial mesh for adaptive refinement can be seen in Fig. 8. It consists of 1 902 triangular elements and provides no boundary layer resolution. The elements are of the same size as the step height.

In Fig. 9, skin friction downstream of the step is plotted on a logarithmic scale. Here, we show results for $p = 1 \ldots 4$ on the structured meshes. The approximation quality of skin friction gets better with both mesh refinement and order increase.

In Fig. 10, two adapted meshes can be seen (with focus on the region downstream of the step). The first one has 15 192 elements and stems from a $p = 1$ adaptation. The second one has 4 396 elements and results from a $p = 3$ adaptation. Both provide a very good boundary layer resolution towards both horizontal and vertical walls, and are clustered towards the corner. The $p = 1$ mesh, however, needs more elements within the recirculation region compared to the $p = 3$ mesh to resolve vortical structures. The Mach number contours on the latter mesh look very smooth (see Fig. 11). Skin friction on both these meshes follows the same trend (see Fig. 12). Although both approximations use almost the same amount of degrees of freedom, the $p = 3$ distribution is considerably smoother than the $p = 1$ distribution. The $p = 3$ is on top of the $p = 4$ solution on the 19 968 element mesh which has ten times the number of degrees of freedom. It even shows more details around $x = 0.1$. 

Figure 6: Skin friction coefficient along the flat plate on adapted meshes ($p = 3$).
Figure 7: Close-up of the meshes for the backward facing step.

(a) $n_e = 4992$

(b) $n_e = 19968$

Figure 8: Close-up of the initial mesh used for adaptation ($n_e = 1902$).
Figure 9: Skin friction coefficient downstream of the backward facing step on the structured meshes.

Figure 10: Adapted meshes for the backward facing step.
Figure 11: Mach contours behind the step.

Figure 12: Skin friction coefficient downstream of the backward facing step on adapted meshes.
C. Turbulent, transonic flow over the RAE 2822 airfoil

Now, we consider turbulent flow around the RAE 2822 airfoil which has been extensively investigated both experimentally\textsuperscript{30} and numerically. Again, we employ the Spalart-Allmaras turbulence model. We will consider specifically test case 9, defined by the following free-stream values: \(Ma = 0.73\), \(\alpha = 3.19^\circ\), \(Re = 6.5 \cdot 10^6\). Following,\textsuperscript{31} we use wind tunnel corrected conditions, i.e. \(Ma = 0.734\), \(\alpha = 2.79^\circ\), \(Re = 6.5 \cdot 10^6\). We prescribe the SA working variable in the free-stream as \(\hat{\nu}_\infty/\nu_\infty = 3\).

Within the high-order workshop,\textsuperscript{32} quartic quadrilateral meshes have been provided. These offer appropriate resolution for both the boundary layer around the airfoil and the shock on the upper side of the airfoil. In the case of curved boundaries, elements sharing these boundaries have to be curved as well. In the case of turbulent flow, where elements close to the wall are highly stretched along the boundary, interior elements have to be curved as well in order to circumvent intersecting element interfaces. At the moment we can only use globally curved meshes which consist of second order triangles (defined by six points). Therefore, we split each quartic quadrilateral (which is defined by 25 points) into 8 quadratic triangles. This way, we can use the whole resolution offered by these meshes. The coarsest of the provided meshes thus consists of 4,048 elements (see Fig. 13). Even on this mesh, \(y^+\) values along the airfoil range between 0.5 and 2.5 for this test case which is sufficiently low. The far-field is located 20 chord lengths away from the airfoil. We employ characteristic upwinding at the far-field boundary. The airfoil itself is modelled as an adiabatic no-slip wall.

![Figure 13: Close-up of the coarsest and finest RAE 2822 mesh](image)

In Fig. 14, Mach number contours around the airfoil are given. The flow accelerates on the upper side until it reaches supersonic levels resulting in a shock. One can clearly observe the interaction of the shock and the very thin boundary-layer. The latter thickens up considerably after being hit by the shock. There is a notable increase in solution quality from \(p = 1\) to \(p = 3\). This is especially visible when comparing the \(p = 3\) solution on the coarser meshes with the \(p = 1\) solution on the finer mesh. Oscillations due to the shock are restricted to the vicinity of the shock.
Figure 14: Mach number contours around the RAE 2822 airfoil
D. Laminar flow around a delta wing

As a three-dimensional example, we consider subsonic, laminar flow around a delta wing with a sloped and sharp leading edge and a blunt trailing edge. The free-stream Mach number is \( \text{Ma} = 0.3 \), the angle of attack \( \alpha = 12.5^\circ \), and the Reynolds number (based on the mean chord length of 1) \( \text{Re} = 4000 \). We use characteristic upwinding at the farfield boundaries. The delta wing is modeled as an isothermal no-slip wall with \( T_w = T_\infty \).

We perform computations on the meshes provided by the high-order workshop\(^{12} \) (\( n_e = 408, 3264, \) and 26112). Please note, that these meshes include degenerated elements (quadrilateral boundary faces with two coinciding vertices and the resulting elements) and zero-measure faces. The first do not pose a problem to our solver. The latter, however, have to be skipped during assembly.

In Fig. 15, Mach number slices of \( p = 2 \) solution on the 26112 element mesh can be seen. One can clearly observe the vortex roll-up along the wing edge. More refinement in the wake region, however, would be necessary to mitigate the dissipation of vortices.

In order to plot the error in lift and drag coefficients, we use the reference values given by Hartmann in the second workshop (\( c_{D,\text{ref}} = 0.1658, c_{L,\text{ref}} = 0.347 \)). Due to the sharp edges, convergence in these functionals is suboptimal (between first and second order). Nonetheless, polynomial degrees higher than 1 seem to pay off in accuracy.

![Mach number slices along the delta wing (\( p = 2, n_e = 26112 \)](image)

Figure 15: Mach number slices along the delta wing (\( p = 2, n_e = 26112 \))
V. Conclusion

We have given an overview of our computational framework with a focus on its implementation. We showed the advantages of using algorithmic differentiation, and generic programming and object orientation. Finally, we applied our framework to laminar and turbulent flow problems in two and three dimensions. Both uniform and adaptive computations have been performed in which the latter have proven superior.

References


