

This is a preprint of the following article, which is available at: <http://mdolab.engin.umich.edu>

Charles A. Mader, Gaetan K. W. Kenway, Anil Yildirim, and Joaquim R. R. A. Martins. ADflow: An Open-Source Computational Fluid Dynamics Solver for Aerodynamic and Multidisciplinary Optimization. *Journal of Aerospace Information Systems*, 2020. doi: 10.2514/1.I010796

The original article may differ from this preprint and is available at:

<https://arc.aiaa.org/doi/10.2514/1.I010796>.

# ADflow: An open-source computational fluid dynamics solver for aerodynamic and multidisciplinary optimization

Charles A. Mader, Gaetan K. W. Kenway, Anil Yildirim, and  
Joaquim R. R. A. Martins

*University of Michigan, Ann Arbor, Michigan, 48109*

## Abstract

Computational fluid dynamics through the solution of the Navier–Stokes equations with turbulence models has become commonplace. However, simply solving these equations is not sufficient to be able to perform efficient design optimization with a flow solver in the loop. This paper discusses the recommendations for developing a flow solver that is suitable for efficient aerodynamic and multidisciplinary design optimization. One of the major recommendations is to be able to load the flow solver as a library that provides direct memory access to the relevant data. Other recommendations are to use a higher-level language for scripting and to pay special attention to solution warm starting, code efficiency, flow solver robustness, and solution failure handling. As an example of a flow solver that follows these recommendation, we present the open-source flow solver ADflow. Results from aerodynamic optimization, aerostructural analysis, and aerostructural optimization using ADflow demonstrate the performance advantages claimed in the recommendations. The publication of these recommendations and the availability of the source code opens the door for other solvers to adopt the same application programming interface. ADflow is part of a wider aerodynamic shape optimization tool suite that is also available under an open-source license.

## 1 Introduction

The increase in computational power and availability has profoundly changed how computational methods are used in engineering design. Computationally intensive simulations that were once used only for final design verifications are now used on a daily basis at the preliminary design stage. This increased power and availability may be exploited in a number of ways:

1. Simulations can be performed with higher resolution—either spatial or temporal.
2. Simulations can be performed with more complex physical modeling—for example solving the Reynolds-Averaged Navier–Stokes (RANS) instead of the Euler equations.
3. More flight conditions can be analyzed for a given geometry.
4. Multiple designs can be evaluated for improvement and for understanding the design performance trades; this may involve parameter sweeps or the use of an optimization algorithm.
5. Computational models that are traditionally treated in a segregated fashion can be integrated to perform a multidisciplinary analysis.

Venkatamaran and Haftka [1] considered the historical effects of increasing computational performance on structural analysis and optimization. They noted that computational analysis tends to follow Parkinson’s Law [2], which states that the work done expands to fill up all the available time. A related law by Thimbleby [3] states that software applications grow to fill up increased computer memory, processing capabilities, and storage space. Venkatamaran and Haftka [1] also point out that anecdotal evidence suggests that time required for “adequate” structural analysis has remained constant, at 6 to 8 hours over the last 30 years. This indicates that computational improvements have been used to refine the computational models, which describes scenarios 1 and 2 above. We believe that the main reason for this is because the first two scenarios alone do not fundamentally change the complexity of an engineering design work flow.

In this work, we examine the requirements necessary to fulfill scenarios 3 through 5 in the context of computational fluid dynamics (CFD). These three scenarios require the computational method to be used repeatedly in a completely automated fashion, which in turn requires additional features for successful computation. One of these required features is the ability to use the solver as a compiled library with direct memory access through a well-defined, streamlined application program interface (API). This allows the solver to be deployed effectively as part of an analysis framework on large-scale high-performance computing (HPC) facilities.

In this paper, we demonstrate how these requirements are met in ADflow, a structured, multi-block, overset flow solver, which is available under an open-source license.<sup>1</sup> Specifically, ADflow is used to solve both aerodynamic and aerostructural design optimization problems for the Common Research Model (CRM) geometry [4]. All of the computations demonstrated in this paper are steady-state RANS solutions. However, the concepts outlined in this paper are not limited to a particular fidelity choice or solution methodology. The API has been applied to 2D and 3D panel solvers as well as multiple 3D flow solvers. ADflow is also capable of time-accurate and time-spectral calculations in addition to the steady-state solutions shown here.

---

<sup>1</sup><https://github.com/mdolab/adflow>, accessed March 2020

The outline of the paper is as follows. Section 2 details the requirements for a multidisciplinary solver and Section 3 introduces the concept of the solver as a software library. Section 4 describes the Python API developed to address the needs listed in Section 2. These sections are meant to be a general guide for solver requirements and are therefore solver agnostic. Section 5 details how these requirements were met for the ADflow solver, and provides a summary of previous studies made possible by ADflow. Finally, Section 6 presents the results from a number of analyses and optimizations to demonstrate the performance of ADflow. Section 7 summarizes the main conclusions of this work.

## 2 Requirements for an efficient multidisciplinary flow solver

The desirable characteristics for a flow solver to be used for multidisciplinary analysis or optimization are different from those of a stand-alone solver. The goal of a stand-alone flow solver is to solve for a given geometry and flow condition with sufficient engineering accuracy as quickly as possible. To solve a multidisciplinary analysis or optimization problem, the flow solver runs as part of a larger framework and needs to be run multiple times in succession without manual intervention. This has several implications for the flow solver requirements, as detailed below.

### 2.1 Solution failure handling

When using a CFD solver within an automatic process, as is necessary for multidisciplinary analysis and optimization, the solver is often required to analyze a wide range of operating points without user intervention. In this scenario, it is likely that the flow solver will be tasked to run one or more analyses that fail to generate an acceptable solution. For a truly automatic procedure, the solver must fail gracefully without incurring an unrecoverable fault. This is particularly important for HPC simulations where the overall process can take several hours or days, and aborted processes incur the additional cost of resubmitting the job and waiting for its turn in the queue.

As a result, an automated process requires good exception handling to be included in the solver. The important cases that need to be handled are when the solution fully converges, diverges, partially converges, stalls, or produces a NaN (not a number) during computation. Most of these cases are simple to handle by monitoring the residual of the flow equations and using logic trees. The various cases can then be handled by returning boolean values to the user or to the driving algorithm upon completion.

The major exception to this is the case where NaNs are encountered. In this case, the solver needs to be reset, including a full re-initialization of the flow to ensure that all of the NaNs in memory are purged so that subsequent flow solutions are not aborted because of a preexisting NaN in memory. While fully resetting the flow solution adds cost to the overall process, it is still far more efficient than re-initializing the flow solver, including the reallocation of all the required memory, as would be required for a stand alone solver.

## 2.2 Solution restart

A second implication from the requirement to run several solutions automatically in sequence is that there is a strong motivation for minimizing the cost of each solution in the sequence. The simplest way to accomplish this is to implement a solution restart procedure, where each solution after the first one starts with the converged state of the previous one. In many cases, for example when computing drag polars, parameters sweeps, and performing gradient-based optimizations, this previous solution state is a better starting point than the default uniform flow. While this can be accomplished with file I/O for most solvers, it is much faster to do through memory. Solution restarts can also be combined with a good choice of algorithm to speed up the subsequent solutions. In particular, Newton’s method yields excellent terminal convergence with a good starting point, a property that can be utilized when the solver is restarted with the previous solution as the initial guess. For cases where this previous solution is not a good starting point, this restart functionality should be made accessible as an option through the API, allowing the user to disable this functionality if starting from a uniform flow is more beneficial.

## 2.3 Robust startup

Having a robust startup method is useful regardless of how a flow solver is being used. However, in our case, where we compute several consecutive solutions in an automated fashion, having a robust startup method is extremely important. The solver needs to be able to converge to a solution from the wide variety of starting points determined by the automated process.

This is also needed every time the flow is reset due to a bad solution because a full startup is required at the subsequent solution point. Furthermore, in the context of design optimization, the optimizer is likely to try infeasible intermediate designs, such as cases that exhibit massive flow separation. While Newton’s method yields good terminal convergence, it usually fails in the early stages of convergence for these extreme cases. These factors increase the need for a robust startup method.

## 2.4 Fine-grained iteration control

When using a flow solver in a coupled analysis (for example, coupled with an external structural solver or a propulsion model) it is also important to be able to control the number of iterations performed for a given solution. In many of these cases, it has been shown that completing partial flow solutions between coupling updates allows the coupled solution to be completed for only a marginal increase in the total flow solution cost. This has been shown for example, in static aeroelastic analysis by Kenway et al. [5], who demonstrated that a relative convergence tolerance of 0.1 per cycle for the flow solver is sufficient to converge the system.



## 2.5 Efficient convergence through all phases of solution

A typical external aerodynamic simulation, such as those we compute with ADflow, can be split into three phases: startup, transition, and terminal. In the startup phase, the initial flow solution interacts with the near-field of the aerodynamic surface. In the transition analysis phase, the flow solver handles the interactions between the near-field solution and the far-field boundaries. In the terminal phase, the solver has already captured the overall flow patterns and converges the numerical solution of the flow to further reduce the residuals to the specified convergence tolerance.

Conventional engineering flow simulations tend to focus on the first two phases. This is because it is only necessary to converge through the transition phase of the solution far enough to have engineering confidence in the solution. However, when performing an optimization, tight numerical convergence of the solution is desirable, especially towards the end of the optimization process, so all three phases of the solution become important.

Different algorithms have different convergence characteristics in each of these phases. Therefore, it becomes important to be able to switch easily between solution algorithms during each simulation to maximize the convergence performance. These switches should be automatic and be based on the relative reduction of the nonlinear residual norm, which is a good metric for monitoring the convergence stage.

## 2.6 Direct memory access and API

The most common approach for handling stand-alone flow solvers in either optimization or multidisciplinary analysis is through file I/O. Using this approach, a unifying framework or driving script is configured to automatically generate input files and parse output files for the various solvers and optimizers in use. While this technique can almost always be used to couple codes together, it has several drawbacks.

The first drawback is the limited availability of disk bandwidth and the associated wall-time required to write, read, and parse the associated data. On massively parallel computing machines, file I/O is typically a shared resource, and therefore, the decrease in throughput experienced by a given user can be significant. The data transfer speeds that can be sustained between computational nodes with a high speed network far exceeds the speeds obtained with file I/O. Using parallel solution techniques for multiple disciplines or optimizers further complicates the information transfer.

The second drawback of the file I/O approach is the potential loss of accuracy. If the analysis output is written using ASCII with a limited number of digits or using binary with single precision (to save disk space and I/O time), information is lost relative to an initial double precision reference when the information is subsequently reloaded. To eliminate this discrepancy, it is necessary to use binary double precision values for all saved information, which leads to high disk usage.

The third drawback is the repeated execution of the same stand-alone code. Each time a code is called, a new process must be started, and one-time initialization functions are typically performed. These operations are typically not as optimized for speed as the remainder of the code. In addition, during optimization and multidisciplinary analysis, successive solu-

tions are often closely related. For iterative methods, it is prudent to reuse this information from one solution to the next to reduce computational cost. This operation requires a restart capability, which is generally not an onerous requirement, but it does increase the amount of information to be written to and read from the disk. These two factors make it significantly more expensive to run many subsequent analyses with a stand alone code as compared to a code run as a library with an API.

Unfortunately, the file I/O approach is the only option if an API is not supplied with direct access to the required functions, as is often the case with commercial codes. To avoid the pitfalls of file I/O, it is critical that all data transfer from the CFD code occur strictly using *direct memory access*. Using this approach, the analysis code is compiled as a library, rather than a stand-alone executable, and a process script is used to direct the sequence of operations performed during the analysis or optimization. The process script then configures each subsequent analysis to run directly, rather than through an input file. This allows all the data that has to be transferred in and out of the CFD solver—the aerodynamic states, forces, and gradients, for example—to be passed through memory. This eliminates the need to write any data to disk, which greatly reduces the cost of cycling iterations. Since we are passing variables through memory, there is no cost to stopping and starting the iteration process, which happens when updating coupling variables or when switching iteration algorithms. Kenway [6] compares the relative cost of direct memory access and file I/O approaches for a static aeroelastic solution with ADflow, where he found that the I/O approach was twice as costly.

## 2.7 Code efficiency

Code efficiency considerations are more important for codes used in multidisciplinary analysis and optimization. This is mostly because the codes run many times to iteratively optimize the design. This iterative process introduces an additional cost multiplier on the already expensive analysis routines. Therefore, it is particularly important to improve the computational efficiency of analysis codes that are used in multidisciplinary design optimization.

To develop efficient code, we consider three levels of efficiency. The first and most important level is algorithmic efficiency, which is achieved by using state-of-the-art algorithms to converge the linear and nonlinear systems of equations that arise during an optimization process. The next level of efficiency comes from having a direct-memory-access API, which eliminates the impact of any file I/O limitations on the performance of the solver, as previously mentioned. The final level of efficiency requires code optimizations that are specific to the algorithms and hardware being used. These optimizations can include reducing memory bandwidth limitations, maximizing vectorization, minimizing cache misses, and other similar improvements.

## 2.8 Additional requirements for efficient multidisciplinary design optimization

While having the capabilities listed in the previous subsections is sufficient to enable efficient multidisciplinary analysis, given the high cost of analyzing most multidisciplinary systems, it is important to use efficient optimization methods as well as efficient solvers when conducting multidisciplinary optimization.

As shown in the study by Yu et al. [7], gradient-based optimization algorithms are much more efficient at finding optimal solutions for CFD-based optimization problems than gradient-free optimization algorithms. To this end, not only is it important to have efficient primal solution algorithms, but to also have efficient computation of derivatives for a multidisciplinary flow solver. In particular, efficient computation of derivatives of a few functions of interest with respect to a large number of design variables is required. The adjoint method is a useful approach for accomplishing this [8–10]. Kenway et al. [11] describes efficient approaches for implementing adjoint methods for CFD solvers and benchmarks ADflow and OpenFOAM adjoint implementations.

## 3 The CFD solver as a library

Many of the requirements listed in the previous section can be achieved by viewing the CFD solver as a library. This approach enables the required level of access to the code using an API while maintaining modularity in terms of code development. Furthermore, a common interface can be developed for multiple CFD codes, enabling the interchangeable use of these CFD solvers as modular components in a broader computational framework.

### 3.1 Code wrapping

To treat the solver as a library and implement the API, it is necessary to wrap its functionality to control it using a scripting language. There are three approaches for providing scripting capability for a solver with increasing levels of intrusiveness:

**File I/O wrapping:** This is the simplest, least intrusive, and most universal of the methods because it can be done by treating the solver as a “black box” without having access to the source code. Using this approach, a script writes an input file, executes the solver, and then parses the resulting output. However, this approach suffers from the drawbacks described previously. The DAfoam wrapper for OpenFOAM developed by He et al. [12] is an example of this approach.

**Function wrapping:** This level of wrapping exposes some but not all of the underlying methods in the solver. This is the approach used to wrap ADflow. For example, methods such as `solve` or `getSolution` are made available through the API, but the lower-level functions used by the solver are not. This method is often employed when the code was written originally as a stand-alone solver and just a subset of high-level methods required for the API are exposed for the scripting level interface.

**Direct object wrapping:** The most intrusive wrapping approach exposes *all* of the underlying data and methods to the scripting interface. The scripting code is responsible for creating all the required objects, down to the lowest level. This approach is most often used when developing a wrapper for an object-oriented code written in C++. An example of a CFD code that uses this approach is elsA [13, 14].

## 3.2 Example workflow using Python

The vast majority of CFD programs rely on either a graphical user interface (GUI) or text user interface (TUI) to control the execution of the solver. It is often the case that a GUI is added on top of an existing TUI, such as the commercial packaging of the OpenFOAM open-source solver [15, 16]. While GUIs help inexperienced users quickly learn the software, they are usually not flexible enough to effectively implement the scenarios 3 through 5 described in Section 1. For these more complex tasks, the ability to quickly and easily script the computational software is a necessity.

The most common way to script TUI-based analysis methods is to use a scripting language to automatically generate an input file, launch the solver, and then parse the resulting text-based output for further analysis. This procedure is tedious and error prone, and output parsing tends to be fragile. A better approach is to perform scripting using the CFD solver directly. Furthermore, with an easy to use yet powerful scripting language such as Python, simple scripts can completely replace the TUI. The use of scripting to control the solver facilitates the transition to the more extensive scripting required for complex tasks.

Figure 1: Example of control script for solving a flow problem.

```
# Import modules
from solverlib import FLOWSolver
from baseclasses import AeroProblem
# Aerodynamic problem description
ap = AeroProblem(name='flow', mach=0.5, alpha=1.0, altitude=0.0, areaRef=1.0, chordRef=1.0)
options = {User Options} # Only non-default options
CFDSolver = FLOWSolver(options=options) # Create solver object
CFDSolver(ap) # Solve problem
```

Figure 1 shows a simple control script for solving a flow problem. This script includes the main settings of a typical TUI file for a CFD solver: flow conditions, normalization values, and solver parameters. The only additional complexity comes from the module imports and the creation of the two required Python objects, `AeroProblem` and `CFDSolver`. This type of run file is functionally equivalent to a TUI file. The power of this approach comes from the flexibility of implementing both simple and complex automation tasks.

Consider, for example, the creation of a drag polar for an airfoil, which requires a sweep over a range of angle of attack variables. Figure 2 details the script that can do this task with Python. The polar requires only a simple `for` loop over the required angle of attack range.

The script writes the results to a simple text file for further processing. In this script, we also take the opportunity to compute a derived value (the lift-to-drag ratio), demonstrating the ability to perform customized post-processing online with the aerodynamic simulations. This example highlights some of the advantages of the pure scripting approach over a scripting language that creates an input file and parses the results: No restart files are written or read, and even though the solver is called multiple times, the initialization needs to be run only once.

Figure 2: Control script for creating a drag polar.

```
# Import modules
from solverlib import FLOWSolver
from baseclasses import AeroProblem
# Aerodynamic problem description
ap = AeroProblem(name='flow', mach=0.5, alpha=1.0, altitude=0.0, areaRef=1.0, chordRef=1.0)
options = {User Options}           # Only non-default options
CFDSolver = FLOWSolver(options=options) # Create solver object
f = open('polar.txt', 'w')
for i in range(0,10,11):
    ap.alpha = i                    # Set new angle of attack
    CFDSolver(ap)                  # Solve problem
    funcs = {}
    CFDSolver.evalFunctions(ap, funcs) # Extract solution
    f.write('%g %g %g %g\n'%(ap.alpha, funcs['cl'], funcs['cd'], funcs['cl']/funcs['cd']))
f.close()
```

## 4 Python API

The key to using the flow solver with a scripting language effectively is a well-designed API. To that end, we have developed a Python API that meets all of the requirements for a solver that is to be used in multidisciplinary analysis and design optimization. This API is extensible to various types of flow solvers and has been demonstrated on several different types of codes, including a structured multi-block and overset solver (ADflow), an unstructured solver (OpenFOAM) [15, 16], a 3D surface panel code (Tripan) [17], and a 2D airfoil solver (XFoil) [18]. The following subsections describe the key elements of this API.

### 4.1 API concept

The fundamental idea driving the development of this API is the concept that in a truly extensible multidisciplinary framework, all of the components must be modular. It is unrealistic to expect that all disciplines in a multidisciplinary analysis to be coded in a monolithic framework. This would limit the ability of the code to be extended to accommodate future

needs. Therefore, we define the boundaries of a typical CFD analysis to establish a general method for modularizing CFD codes.

The key concept for enabling this is to define the geometric surface of the CFD problem as the point of interaction for the flow solver. In most CFD problems, this geometric surface defines the boundary of the flow domain. This is true regardless of the flow solver fidelity level. Both analyses with a volumetric analysis domain, such as RANS and Euler CFD codes, and analyses with a surface domain, such as a panel code, can be handled using this approach.

Furthermore, having the interface defined at the surface allows for straightforward use in both multidisciplinary analysis and design optimization applications. It is on this surface that physical quantities are integrated. For example, the transfers of the heat fluxes in an aerothermodynamic analyses or the displacements and forces in an aerostructural analyses are done through this surface.

A second important concept for the API is the separation between the flow conditions definition for a given analysis and the geometric definition of the problem. Several tasks, from parameters sweeps to multipoint optimization problems, require the analysis of a single geometry at multiple flow conditions. By separating the definition of the from the solver itself, it is possible to analyze any number of these flow conditions without re-initializing the flow solver and incurring the associated startup penalty.

## 4.2 API layout

Using the concepts mentioned above, the API needs to have the ability to:

- Manipulate the surface of the CFD geometry
- Specify the flow conditions
- Solve for the flow state variables
- Evaluate the functions of interest
- Recover the solution from a failure state
- Evaluate the solver derivatives

Here, we elaborate on each of these requirements. In particular, we detail the specific implementation we have developed for the API and how each of the specified requirements is met through the API functionality. Figures 3 and 4 show simplified UML diagrams for the solver and aerodynamic problem classes that embody the API outlined here. The figures are simplified by leaving out some of the detailed private attributes and functions that are solver specific and not part of the general API. The basic API layout is composed of a subset of methods in these figures that provide the essential functionality.

### 4.2.1 ADflow class layout

The ADflow API uses class inheritance, as shown in Figure 3, where each class inherits the properties and methods of all of the classes to its left. The base class is the Python `object` class, which is part of the Python standard and is the basic building block for all classes in this language.

The `BaseSolver` class is used for different types of solvers and defines methods for option handling and class naming, which are common to all the solvers we implement. The `AeroSolver` class is the first layer of specialization for aerodynamic solvers. This class contains attributes to access mesh and geometry objects, as well as basic implementations of most of the API calls outlined in this work. The fourth and final class is the `ADFLOW` class, which contains specific implementations of the functionality described in this work.

The purpose of each of these calls is provided in the following sections. Functions starting and ending with `__` are intrinsic Python functions that are part of a standard Python class definition.

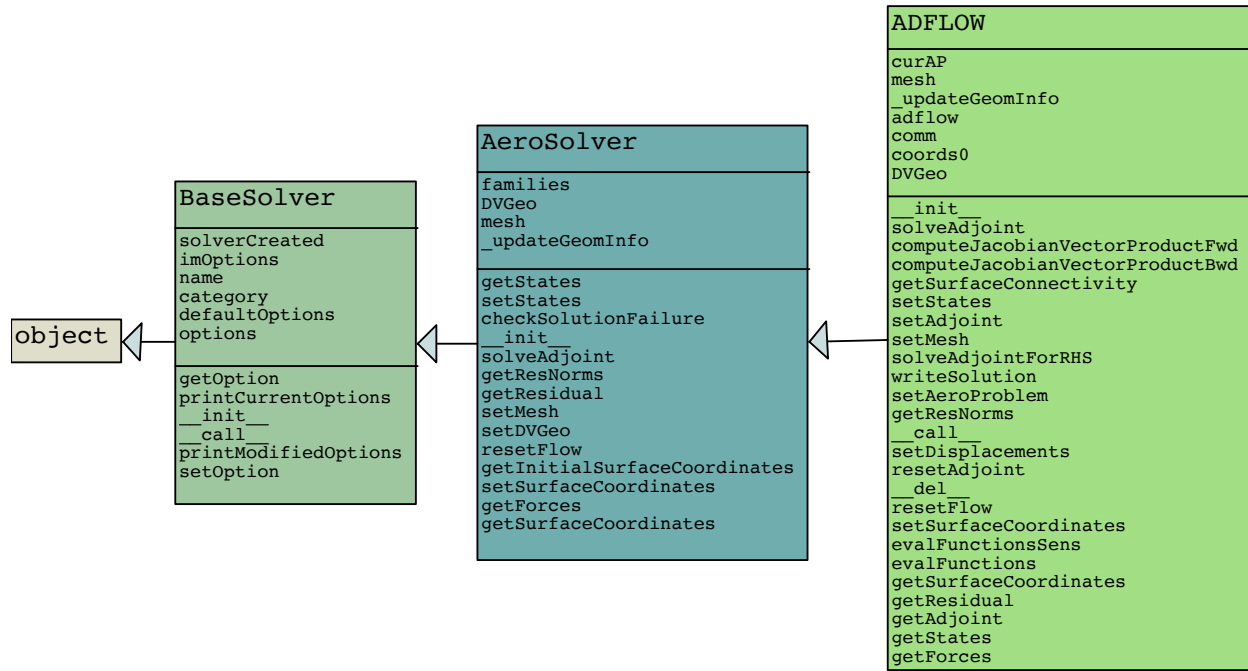


Figure 3: Simplified UML diagram of ADflow and its base classes.

### 4.2.2 AeroProblem class layout

The `AeroProblem` class (shown in Figure) 4 stores and updates all of the information required to run an aerodynamic solution at a given flow condition. This includes functions to treat these variables as design variables and to generate a complete thermodynamic state from various combinations of input data.

This class contains an instance of the `ICAOAtmosphere` class in the `atm` attribute. This class has a smoothed implementation of the ICAO standard atmosphere tables that computes fluid temperature, pressure, and density for the altitude corresponding to the flight condition.

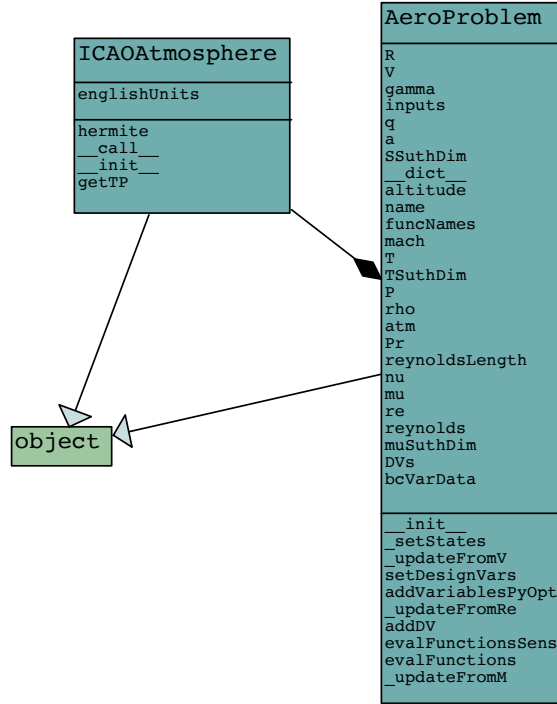


Figure 4: Simplified UML of the aerodynamic problem class.

### 4.2.3 Surface manipulation

There are three main functions required for the manipulation of the boundary surfaces of a CFD problem, whose names are self explanatory: `getSurfaceCoordinates`, `setSurfaceCoordinates`, and `getSurfaceConnectivity`. As previously mentioned, the philosophy of this API is that these boundary surfaces represent the interface between the CFD solver and other components or disciplines in a multidisciplinary analysis.

This approach allows the API to be used for both 3D volume mesh codes, such as those based on the RANS or Euler equations, or lower-fidelity codes, such as panel codes. However, this means that any mesh manipulation tasks, such as mesh warping, mesh regeneration, or mesh adaptation for volume meshes must be handled inside the flow solver definition. This can be accomplished in many different ways and is solver specific. Therefore, we do not attempt to prescribe an approach to handling volume meshes in this API. In ADflow, the volume mesh is handled by plugging an additional Python module into the flow solver at the Python layer, as shown in Figure 7, allowing different mesh manipulation tools to be used



as needed.

Figure 5 shows the `getSurfaceCoordinates` function, which returns the coordinates of CFD boundary surfaces. The default functionality is to return all solid wall boundaries of the model, while the `groupName` argument allows the user to select specific subsets of the boundary points to be returned.

Subset-selection is important for some types of multidisciplinary analysis. For example, in a static aeroelastic (aerostructural) analysis with a wing-body-tail CFD mesh that only has a wing-box structure, the user would probably not want the deflections of the wing structure to affect the fuselage or the tail. With this API, the user can request just the coordinates of the wing surface, so that this subset can be used to create the association between the aerodynamic and structural meshes. The surfaces are typically stored in a distributed manner, with a portion of the surface on each processor, eliminating serial processing bottlenecks.

Figure 5: Function that returns the surface coordinates that define the boundary surface of the flow problem.

```
def getSurfaceCoordinates(self, groupName=None):
    """
    Return the coordinates for the surfaces defined by groupName.
    """
    return coords
```

The `getSurfaceConnectivity` function returns a connectivity array for the surface coordinates. This connectivity describes the boundary surface mesh of the CFD based on the coordinates returned in the `getSurfaceCoordinates` function. This additional information is required to facilitate the communication with other disciplines, such as structural analysis and mesh deformation.

The final surface manipulation function is `setSurfaceCoordinates`, which allows the coordinates, as returned in `getSurfaceCoordinates` to be updated at any time.

#### 4.2.4 Set flow conditions

The function that sets the flow conditions is internal to the solver class and is not part of the API. The flow condition information is contained in an `AeroProblem` class. This class allows the user to specify the required flow conditions in a variety of ways. For external flow calculations, the class computes the full set of thermodynamic variables required by the flow solver. Additionally, specific boundary conditions with specified flow properties can be set for boundaries, such as inflow or outflow conditions. Any number of these problems can be setup and passed to the solver for sequential solutions.

#### 4.2.5 Solve flow problem

The core solver function is in the `__call__` method, whose signature is:

```
def __call__(self, aeroProblem):
```

This function takes in an `AeroProblem` object and updates any solver specific settings for the information contained in the `AeroProblem`. It also updates the volume mesh based on the current surface, configures the solver with the current options, and handles the file input and output.

This function can be configured to run for a fixed number of iterations, a fixed wall time, or until the solver reaches a specific convergence tolerance. This allows for fine-grained control over the flow solution process, which is useful for optimizations and multidisciplinary analyses, as previously mentioned.

#### 4.2.6 Evaluate quantities of interest

The `evalFunctions` method evaluates the flow solution for quantities of interest, and has the following signature:

```
def evalFunctions(self, aeroProblem, funcs, evalFuncs=None):
```

The incoming `AeroProblem` identifies the flow solution to be used, while `evalFuncs` is a list of the functions to be evaluated. The evaluated functions are added to the `funcs` dictionary, which can be accessed from the main script. By using a dictionary, we identify the different functions evaluated from separate `AeroProblems` with unique keys, making it substantially easier to handle complex cases with many functions and flow solutions.

#### 4.2.7 Recover from a failed solution

Being able to recover from a failed solution is critical for any automated flow solution process. As mentioned previously, it is impossible to guarantee that every flow solution in an automated process will finish successfully.

Fully converged and partially converged solutions are considered successful solutions. These are picked up by the logic tree in the solver and return `FailedSolution = False`. A solution is considered partially converged if it terminates within a predefined tolerance of the target convergence.

Solutions that terminate due to the maximum iteration limit without diverging and without successfully converging are considered *stalled* solutions. Stalled solutions are considered solution failures, so they return `FailedSolution = True`, but because the solution does not contain any NaN values, it does not require any special treatment.

A solution is considered *diverged* if the norm of the total residual increases beyond a certain threshold. Diverged solutions are also considered to be failed solutions and return `FailedSolution = True`. On its own, a diverged solution is not an issue and does not need any special treatment. However, diverged solutions in the CFD solver often trigger an NaN, requiring the flow solution to be re-initialized.

If the solution process produces a NaN, the automated process is interrupted and cannot continue unless all of the NaN's are eliminated from the set of variables that define the state

of the system, both in Python and in the compiled library. In ADflow, this re-initialization is accomplished by the `resetFlow` routine, which purges and resets all of the variables in memory both in the Python layer and in the compiled library. This method has the signature:

```
def resetFlow(self, aeroProblem):
```

where `aeroProblem` corresponds to the flow condition associated with the failed solution.

#### 4.2.8 Evaluate the solver derivatives

As we mentioned in Section 2.8, when running optimizations using expensive computations, such as those inherent in CFD based optimization, it is strongly recommended to use gradient-based optimization methods. To use these methods, we need to compute the derivatives of the objective function and all the constraints with respect to all the design variables in an efficient manner. Therefore, the definition of the API includes methods for computing the solver derivatives.

The standard function for users who just want the aerodynamic derivatives for design is the high-level function `evalFunctionsSens()`. This function handles the entire derivative computation process and updates a dictionary provided by the user with the computed values. For `evalFunctionsSens` to compute the total derivative of the functions of interest, we must be able to compute the entire derivative chain back to the design variables. The derivatives in ADflow are computed using the adjoint method, as described by Kenway et al. [11].

### 4.3 Advanced API layout

A significant amount of functionality can be implemented using the basic API capability. However, with an additional set of functions, we can enable finer grained inter-connectivity in frameworks, such as OpenMDAO [19], which uses the modular analysis and unified derivatives (MAUD) architecture [20]. Specifically, we need functions to:

- Get and set the solver state variables
- Evaluate the solver residuals
- Get and set the adjoint variables
- Get and set coupling variables
- Evaluate matrix-vector products with the state Jacobian and its transpose
- Solve the adjoint equation with arbitrary right-hand sides

The advanced API methods are listed in Figures 3 and 4, and are detailed below.

### 4.3.1 Get and set states

The `getStates` and `setStates` functions are useful for cases where multiple flow solutions are required. The function signatures for these methods are:

```
def getStates(self):  
    return states
```

```
def setStates(self, states):
```

This functionality serves two purposes. First, it allows the state of the system to be saved from one flow solution to the next for a straight-forward restart process. This is particularly important when running multiple cases, such as in multipoint optimization, where the flow solver must regularly switch between flow cases. Second, this functionality allows for tighter integration with frameworks such as OpenMDAO [19]. By providing access to the flow states, the solver can be integrated into various different coupled architectures.

### 4.3.2 Evaluate residuals

Direct evaluation of the flow residuals and their norms is another API functionality that helps with coupled integration. The signatures for these calls are:

```
def getResidual(self, aeroProblem):  
    return res
```

```
def getResNorms(self):  
    return totalr0, totalrstart, totalrfinal
```

These functions evaluate the solver residuals and their norm at the Python level and can help with the implementation of various multidisciplinary solver architectures.

### 4.3.3 Get and set adjoint states

Getting and setting the adjoint states plays the same role for the adjoint solution as getting and setting the primal states for the primal solution. It allows for quicker restarts when multiple adjoint solutions are required and it also allows for a more seamless integration of the adjoint system into frameworks that compute coupled derivatives, such as OpenMDAO [19].

### 4.3.4 Get and set coupling variables

Getting and setting coupling variables is essential for coupled analysis. While this can be done through file I/O, it is far more efficient to do it through the direct-memory-access API. However, the specific nature of the coupling dictates that there needs to be a variety of functions to accomplish this goal. In the cases we have dealt with, these functions include functions to access the distributed forces on the surface, to update the surface shape based on displacements, and to update boundary conditions based on propulsion variables. These functions have the same general form as the “get” and “set” state functions: they either take in or return a vector of coupling variables.

### 4.3.5 Evaluate matrix-vector products with the state Jacobian

Being able to compute matrix-vector products with the partial derivatives of the functions and residuals in the flow solver from the API is extremely useful. In this context, we mean partial derivatives to be the derivatives of these quantities without re-solving the non linear system. These matrix-vector products can be used in adjoint solvers [11] and Newton-Krylov solvers [21], so being able to evaluate these derivatives through the API allows tremendous flexibility when setting up single discipline or multidisciplinary solvers.

The lower-level API functions used for direct interaction with the aerodynamic partial derivative computations are the `computeJacobianVectorProductFwd()` and `computeJacobianVectorProductBwd()` functions, which are used to evaluate vector products with the state Jacobian matrix, and its transpose respectively. These functions allow the user to compute any combination of partial derivatives through the solver, depending on what input arguments are provided.

Figure 6 shows the specific implementation of the backward variant, which computes the transpose vector products used for adjoint derivative computation. This allows derivatives to be computed for any combination of algorithmic differentiation seeds provided and can either be used to compute stand-alone derivative vectors or linear combinations of those derivatives as necessary.

Figure 6: Function for computing the Jacobian-vector products in reverse mode

```
def computeJacobianVectorProductBwd(self, resBar=None, funcsBar=None, fBar=None,
                                     wDeriv=None, xVDeriv=None, xSDeriv=None,
                                     xDvDeriv=None, xDvDerivAero=None):

    """This the main Python gateway for producing reverse mode Jacobian
    vector products. It is not generally called by the user by
    rather internally or from another solver. A mesh object must
    be present for the xSDeriv=True flag and a mesh and DVGeo
    object must be present for xDvDeriv=True flag. Note that more
    than one of the specified return flags may be specified. If
    more than one return is specified, the order of return is :
    (wDeriv, xVDeriv, XsDeriv, xDvDeriv, dXdvDerivAero).
```

### 4.3.6 Evaluate adjoint solver with arbitrary right-hand sides

The final functionality in the advanced API is the ability to solve the adjoint linear system for arbitrary right-hand-side vectors. The core benefit of the adjoint method is its ability to compute gradients efficiently with respect to a small number of outputs. It is, therefore, important to minimize the number of output functions that require an adjoint solution. In multidisciplinary systems, the functions of interest are typically multidisciplinary as well. It is more efficient to compute the combined multidisciplinary right-hand side for these functions and solve a single adjoint system, rather than having to compute the adjoint solution

for each disciplinary output individually. This requires the API to handle communication of arbitrary right-hand side vectors to the adjoint solver. Frameworks, such as OpenM-DAO [19], also benefit from this functionality, since they will often create their own right hand side combinations for the adjoint solver when that functionality is supported.

## 4.4 Plug-in API layout

In addition to the core functionality of the flow solver, the API defines a pair of plug-in handles for specific cases. These plug-in extensions are set using the `setMesh` and `setDVGeo` methods:

```
def setMesh(self, mesh):
    self.mesh = mesh
```

```
def setDVGeo(self, DVGeo):
    self.DVGeo = DVGeo
```

For volumetric flow solvers, such as ADflow, the `setMesh` extension allows a mesh manipulation object to be included in the flow solver. This external object is used inside the flow solver to translate the surface perturbations set in the `setSurfaceCoordinates` call in the main API to the volume mesh defined in the flow solver. It is also responsible for computing the sensitivity of this operation during the derivative process. For surface based solvers, such as panel codes, this extension to the API is not necessary.

If physically meaningful shape variables are desired for design perturbation, an external geometry manipulation module is required to define the perturbations to the surface mesh as a function of these variables [22]. This functionality is added to the flow solver through the `setDVGeo` extension. Again, this external object is responsible for providing both the relationship between the geometric design variables and the surface mesh coordinates as well as the derivatives of these operations. Objects that satisfy these needs in conjunction with ADflow are part of the broader MACH-Aero framework, which integrates all the components required to perform aerodynamic shape optimization <sup>2</sup>.

## 5 ADflow: A CFD solver for multidisciplinary analysis and optimization

The concepts and requirements detailed so far have been general, solver-independent ideas, intended to serve as a guide for solver development. However, specific details and examples are helpful for a successful implementation, so here we describe how the requirements for multidisciplinary design analysis and optimization were met in ADflow. The source code for ADflow is distributed under the GNU Lesser General Public License (LGPL), version 2.1, and is maintained in a GitHub repository. <sup>3</sup> This section starts with a brief introduction to

<sup>2</sup><https://github.com/mdolab/mach-aero>, accessed March 2020

<sup>3</sup><https://github.com/mdolab/adflow>, accessed March 2020

the computational models in ADflow. After detailing how the requirements are met, we list the various studies that ADflow has made possible in the last few years.

## 5.1 Computational model

ADflow can solve the Euler, laminar Navier–Stokes, and RANS equations on structured, multi-block and overset meshes using a second-order-accurate finite-volume approach for the spatial discretization. The inviscid flux calculations can be done with either a Jameson–Schmidt–Turkel scheme using scalar dissipation [23], a matrix dissipation scheme based on the work of Turkel and Vatsa [24], or a monotone upstream-centered scheme for conservation laws (MUSCL) based on the work of van Leer [25] and Roe [26]. The viscous flux calculations use the Green–Gauss approach.

ADflow contains a number of turbulence models for the RANS simulations, including Spalart–Allmaras (SA) [27], Wilcox  $k-\omega$  [28], and Menter shear stress transport [29]. The default turbulence model in ADflow is the SA model, which is fully differentiated for gradient computations with the adjoint method [11]. ADflow can use different variants of the SA model as defined in NASA Turbulence Modeling Resource [30]; we use primarily the SA-noft2 variant.

## 5.2 Enabling multidisciplinary design analysis and optimization

The main enabler for multidisciplinary analysis and optimization in ADflow is the Python API, which has the characteristics described in the previous section. Using this API, we can couple ADflow to other analysis codes efficiently using direct memory access. Furthermore, we can easily obtain the desired solver behavior while recovering from failed solutions or solver restarts between optimization iterations.

These functionalities introduced by the API address many of the challenges we listed in the previous sections. Figure 7 shows the general concept of the ADflow API and how it interacts with other components in a multidisciplinary context. In this figure, the rounded-edge blocks are required Python objects, while the hexagonal blocks are optional Python objects. The regular rectangles are data passed by function calls. Items connected by the lighter arrows are optional, while the others are required. Thus, the most basic analysis possible includes an aerodynamic problem and the core solver, and computes values for integrated quantities such as  $C_L$  and  $C_D$ . More complicated analyses build up from this.

Another important feature of ADflow is the nonlinear solution algorithm. The convergence for steady-state simulations is determined by monitoring the  $L_2$  norm of the residual vector. ADflow computes this norm before each nonlinear iteration and selects the desired nonlinear solver scheme based on the relative convergence metric, which is given by dividing the current residual norm by the initial residual norm that is calculated with free stream conditions. This provides flexibility in the nonlinear solver schemes and the code can select the best solver algorithm based on the convergence stage.

This computation is also performed for subsequent solutions during the optimization process, but the solver uses the initial residual norm for relative convergence calculations as

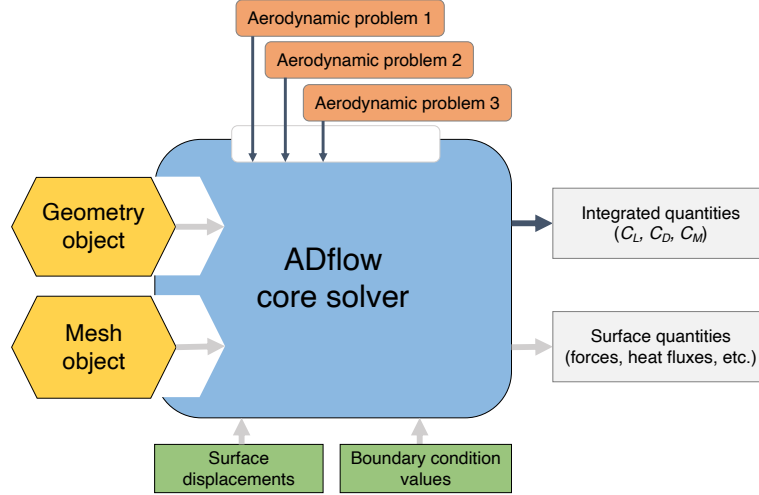


Figure 7: ADflow integration with other components and disciplines in a multidisciplinary context.

the reference. This enables the solver to determine the convergence stage even when we use the previous converged state as the initial guess.

For the initial stages of convergence, we have two alternative algorithms: multi-grid, and approximate Newton–Krylov (ANK). The multi-grid algorithms in ADflow can be used with multi-block meshes, where obtaining coarser levels of the mesh is straightforward for meshes with the correct number of nodes or cells. Using this approach, ADflow can use a 5-stage 4<sup>th</sup> order accurate Runge–Kutta or the D3ADI [31] schemes as smoothers in the multi-grid startup process.

The ANK solver was developed to add robustness to the pure NK algorithm [21]. It uses a pseudo-transient continuation (PTC) method and an approximate Jacobian with the backward Euler time-stepping scheme. This solver does not require coarser levels of the mesh and it is therefore applicable to both multi-block and overset meshes. The approximate nature of the linear system used in the solver, along with PTC, allows the algorithm to progress the solution even when the state is far away from the final solution. The adaptive nature of our implementation allows the solver to reduce the amount of approximation in the linear approximation as the solver converges. This allows the solver to improve in performance as the solution gets closer to the converged state.

When tuning an ANK solver, there is a trade-off between efficiency and robustness. We have tuned the ANK solver defaults to favor robustness. This is because in an optimization context, the optimizer is likely to try infeasible intermediate designs, and also because an interruption of the optimization process is costly. The robustness of the ANK solver enables ADflow to obtain steady-state solutions even with these intermediate cases, which helps the optimization convergence by reducing the number of failed flow solutions.

For the terminal stage of convergence, ADflow switches to the Newton–Krylov (NK) solver. This solver uses Newton’s method to converge the nonlinear system and a Krylov



subspace solver to solve the resulting linear systems. This approach can yield convergence approaching quadratic, but only if the initial guess is in the basin of attraction of the solution. Therefore, we only use this method when the relative convergence of the more robust nonlinear solver is below  $10^{-3} - 10^{-5}$ .

Efficient solver restarting is important within an optimization context, where the flow solver is repeatedly called to solve similar problems between optimization iterations. During successive CFD simulations, we use the converged solution from the previous optimization iteration as the initial guess. If the design changes are large, the nonlinear residual norm increases, and the solver defaults to one of the desired startup strategies. This is done to prevent failures that might occur with the NK solver, when the initial guess is far from the solution. However, if the design changes are small (as it is likely to happen during the final stages of an optimization process), the previous flow solution provides a good enough initial guess for the NK solver to converge. As a result, ADflow can rapidly obtain solutions for new problems with slightly perturbed designs.

When using gradient-based optimization, the flow solver needs to provide the derivatives of the functions of interest (objective and constraint functions) with respect to the design variables. In aerodynamic design optimization problems of interest, there are usually far more design variables than functions of interest. As a result, the derivatives can be efficiently computed using the adjoint method.

Kenway et al. [11] detail the adjoint solver implementation in ADflow. The overall approach is to use automatic differentiation to compute the terms necessary to form the discrete adjoint equations, resulting in accurate derivatives. This approach to adjoint development also reduces the overhead to maintaining the adjoint code, since the automatic differentiation tool can be used to update the derivative code whenever changes are made in the analysis code. Furthermore, the cost of the adjoint approach is independent of the number of variables (but it scales with the number functions of interest), which makes it suitable for solving large-scale aerodynamic shape optimization problems.

For computational efficiency, ADflow implements the three levels of improvements mentioned previously. First of all, we use state-of-the-art algorithms to converge the resulting nonlinear and linear systems. The ANK, NK, and adjoint solvers use Jacobian-free methods to solve the underlying linear solution algorithms. This minimizes the code memory requirements, while the solution algorithms themselves provide fast convergence for the nonlinear and linear systems. Secondly, we have direct memory access between ADflow and other analysis code we couple to it. This removes any file I/O bottlenecks. The flow solver is only initialized once and the allocated memory is recycled only between design iterations. Finally, ADflow uses a cache-blocking technique to minimize cache misses with the residual calculations. Besides mitigating the memory access bottleneck, this also enables us to take full advantage of the vector instruction sets in modern processor architectures. All these enhancements contribute to the performance of ADflow and help reduce the cost of the optimization problems to manageable levels.

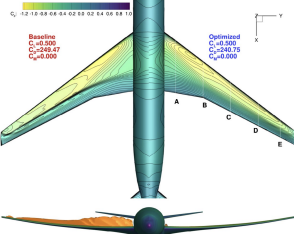
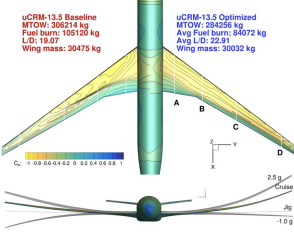
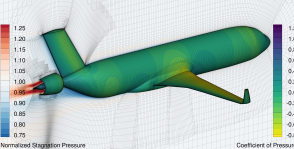
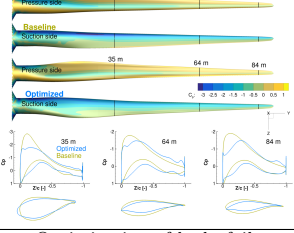
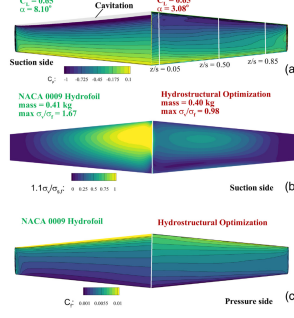
In addition to these enhancements, various implementation details in ADflow help developers to easily extend the code for novel applications. Because the API is written in

Python, developers can use the flexibility of this object-oriented language to achieve the desired results with minimal coding effort. On the other hand, the high-performance routines in ADflow are written in Fortran 90. This enables the developers to use a compiled coding language for parts of the implementation that are performance critical. Furthermore, this Fortran layer is coded in a modular way, so developers can easily implement new turbulence models or modify the governing equations without needing to change the core code. Finally, we use the portable, extensible toolkit for scientific computation (PETSc) as the underlying linear algebra package [32]. This provides us with state-of-the-art implementations of modern linear algebra algorithms, which we rely on for the nonlinear and linear solvers in ADflow. These factors lower the initial coding investment when implementing new features in ADflow and enable users to extend the code for their multidisciplinary applications.

### **5.3 Previous results obtained with ADflow**

ADflow is the core CFD solver in the multidisciplinary design optimization of aircraft with high-fidelity (MACH) framework [5]. We have also used the ADflow API to develop an OpenMDAO wrapper. We have used these two frameworks to solve the design optimization problems listed in Table 1.

Table 1: Optimization problems solved with ADflow.

Application	References
<p><b>Aerodynamic shape optimization</b></p>  <p>Baseline  <math>C_L = 0.398</math>  <math>C_D = 0.047</math>  <math>C_{D0} = 0.009</math></p> <p>Optimized  <math>C_L = 0.398</math>  <math>C_D = 0.039</math>  <math>C_{D0} = 0.007</math></p>	<p>2D transonic aerodynamic shape optimization [33, 34]  3D transonic aerodynamic shape optimization [35–38]  Optimization of novel configurations [39–41]  Formulation of buffet constraint for wing design optimization [42]  2D and 3D supersonic aerodynamic shape optimization [43]  Optimization with spatial integration constraints [44, 45]  Simultaneous design optimization of shape, trajectory, and aircraft allocation [46]</p>
<p><b>Aerostructural design optimization</b></p>  <p>uCRM-13.5 Baseline  MTOW: 306214 kg  Fuel burn: 155120 kg  L/D: 19.37  Wing mass: 30475 kg</p> <p>uCRM-13.5 Optimized  MTOW: 304250 kg  Avg Fuel burn: 84072 kg  Wing L/D: 22.31  Wing mass: 30032 kg</p>	<p>Optimization of a transport configuration [47–49]  Optimization with tow-steered composite structures [50, 51]  Optimization of morphing trailing edge device [52, 53]  Optimization with flutter constraints [54–56]</p>
<p><b>Aeropropulsive design optimization</b></p>  <p>Normalized Stagnation Pressure</p> <p>Coefficient of Pressure</p>	<p>Boundary layer ingestion modeling [57]  Design optimization of a boundary layer ingestion system [58–61]</p>
<p><b>Design optimization of wind turbines</b></p>  <p>Pressure side</p> <p>Baseline</p> <p>Optimized</p> <p>35 m</p> <p>64 m</p> <p>84 m</p> <p>Optimized</p> <p>Baseline</p> <p>35 m</p> <p>64 m</p> <p>84 m</p> <p>Optimized</p> <p>Baseline</p> <p>35 m</p> <p>64 m</p> <p>84 m</p>	<p>Aerodynamic shape optimization of wind turbine blades [62, 63]</p>
<p><b>Optimization of hydrofoils</b></p>  <p>NACA 0009 Hydrofoil  <math>C_L = 0.0362</math>  <math>C_D = 0.005</math>  <math>\alpha = 8.10^\circ</math></p> <p>Hydrostructural Optimization  <math>C_L = 0.0321</math>  <math>C_D = 0.005</math>  <math>\alpha = 3.98^\circ</math></p> <p>Hydrodynamic hydrofoil shape optimization [64]  Hydrostructural optimization of metallic and composite hydrofoils [65–67]</p>	

We have used ADflow for both 2D and 3D aerodynamic shape optimization studies. Li et al. [33] developed a data-based approach for analysis and optimization of airfoil shapes

that resulted in the online airfoil analysis and design optimization tool Webfoil.<sup>4</sup> Mangano and Martins [43] performed multipoint shape optimization of airfoils with mixed transonic and supersonic conditions. He et al. [34] developed a robust framework for aerodynamic shape optimization and demonstrated it for an airfoil shape optimization that started from a circle shape and converged to a supercritical airfoil.

Besides these 2D results, we have performed various investigations based on the NASA Common Research Model (CRM) [68] for both aerodynamic and aerostructural shape optimization. These include single- and multipoint optimizations of the CRM wing [35, 37], and the aerodynamic shape optimization of the CRM wing-body-tail configuration [38, 42]. Some of these cases are open benchmarks developed by the AIAA Aerodynamic Design Optimization Discussion Group. While these efforts focused only on aerodynamics, they demonstrate that ADflow can be used in design optimization with geometry manipulation and mesh deformation algorithms.

We used ADflow along with Toolkit for the Analysis of Composite Structures (TACS) [69] in the MACH framework to perform aerostructural design optimization. These include multipoint [47] and multi-mission [48] optimizations of the CRM configuration that resulted in the development of open benchmark cases for aerostructural design optimization studies, including a higher aspect ratio version of the CRM configuration [49]. These studies demonstrate the effectiveness of ADflow within a multidisciplinary analysis and optimization framework that considers both aerodynamics and structures.

Using MACH, we have also applied our methodology to the design optimization of aircraft utilizing new technologies, such as tow-steered composite wings [50, 51] and morphing wing technology [36, 52, 53]. ADflow has been used in design optimization of novel configurations, including flying wings [39], blended-wing-body aircraft [40], and the D8 configuration [70]. The overset implementation in ADflow [71] enabled us to create a component based aerodynamic shape optimization framework [72], which was used to perform the design optimization of a strut-braced wing configuration [41].

We have also coupled ADflow to pyCycle [73] using the OpenMDAO framework [19] to perform aeropropulsive design optimization [59]. This coupling is important for studying boundary layer ingestion concepts because they require the simultaneous consideration of aerodynamics and propulsion [57]. This framework was used to optimize the design of the STARC-ABL concept [58–61].

We have extended the MACH framework to handle spatial integration constraints [44, 45] and we are currently developing a time-spectral formulation for flutter prediction [54–56]. Furthermore, we are improving the OpenMDAO integration of ADflow to achieve even more flexibility to include other disciplines in the optimization formulations, such as a combined design-allocation optimization problem [46].

Beyond the aircraft applications cited above, we have used ADflow in design optimization of wind-turbines [62, 63] and hydrofoils [66, 67]. One set of baseline and optimized hydrofoils was built and validated in a water tunnel, yielding good agreement with the numerical predictions [65]. Even though ADflow solves the compressible flow equations, the design

---

<sup>4</sup><http://webfoil.engin.umich.edu>

optimization capabilities of ADflow made it a good choice for these incompressible flow design problems.

All of this work was made possible by the considerations we listed in the previous subsection. The flexibility of the API enabled us to couple ADflow to different frameworks and disciplines, while the efficiency of the code reduced the cost of these massive problems.

## 6 Computational performance

While the performance of a multidisciplinary analysis and optimization setup depends on many factors, we focus on the impact of treating the flow solver as a library using three problems. These problems also serve as additional example problems that demonstrate the flexibility of the API.

The first problem is a simple aerodynamic optimization with two design variables, which gives an idea of the relative importance of different phases of the solutions process in an aerodynamic optimization. The second problem is a simple aerostructural analysis, which highlights the additional areas of a multidisciplinary analysis that become performance critical—more specifically, the reduced cost of the flow solution process relative to other portions of the analysis. Finally, in the third problem, we solve an aerostructural optimization problem with the same two design variables. This case extends the multidisciplinary analysis comparison to a full optimization and further demonstrates how the API developed in this work addresses those challenges.

All three cases are based on the CRM aircraft configuration that consists of wing, fuselage, and horizontal tail. The results are reported in TauBench work units (TWU) as defined by the guidelines from the International Workshop on High-Order CFD Methods <sup>5</sup>. This allows for a normalized comparison of the results between different computers and codes, rather than relying on a more subjective iteration count or wall time comparison.

### 6.1 TauBench reference

To benchmark the cost of the computations in this work, we use the TauBench reference test. As outlined in the guidelines, we have run TauBench three times with the command: `mpirun -np 1 ./TauBench -n 250000 -s 10`. This yields a processor TWU time of 6.238 seconds as a reference when running in the NASA Pleiades Cluster with the Ivy Bridge nodes. The detailed results are shown in Table 2.

### 6.2 Common Research Model

We use the CRM geometry for the aerodynamic optimization and the aerostructural analysis. This is a wing-body-tail geometry that is representative of a large commercial transport aircraft [68]. We have selected this case because it provides a realistic example of optimizations that can be run with ADflow. The aerodynamic model is based on previous work by Chen

---

<sup>5</sup><https://www.grc.nasa.gov/hiocfd/guidelines/>, accessed March 2020

Table 2: TauBench work unit (TWU) results for the NASA Pleiades Cluster with the Ivy Bridge nodes

Run	Computation time	Communication time	Ratio	Total time
1	6.189	0.022	0.004	6.211
2	6.217	0.026	0.004	6.243
3	6.233	0.026	0.004	6.259
Average	6.213	0.025	0.004	6.238

et al. [38], while the aerostructural model is the undeformed CRM benchmark developed by Brooks et al. [49].

Figure 8 shows the CFD surface and structural models, as well as the solution at the aerostructural optimum. The full family of CFD meshes for the CRM is detailed in Table 3. We use a wing-box mesh with 25 998 third-order shell elements based on 101 129 nodes with 606 774 degrees of freedom. Mesh convergence results for the full family of CFD meshes are shown in Table 4.

Table 3: CRM mesh data

Level	Number of cells	Off-wall spacing	$y+$ max.
L0	47,372,288	$6.0 \times 10^{-5} - 1.2 \times 10^{-4}$	0.3597
L0.5	14,233,600	$9.0 \times 10^{-5} - 1.8 \times 10^{-4}$	0.5414
L1	5,921,536	$1.5 \times 10^{-4} - 2.4 \times 10^{-4}$	0.7848
L1.5	1,779,200	$1.8 \times 10^{-4} - 3.6 \times 10^{-4}$	1.2659
L2	740,192	$3.0 \times 10^{-4} - 4.8 \times 10^{-4}$	1.9083

Table 4: CRM mesh convergence

Level	$C_L$	Aerodynamic $C_D$	Aerostructural $C_D$
L0	0.500	0.0217	0.0225
L0.5	0.500	0.0219	0.0227
L1	0.500	0.0223	0.0231
L1.5	0.500	0.0237	0.0245
L2	0.500	0.0258	0.0266

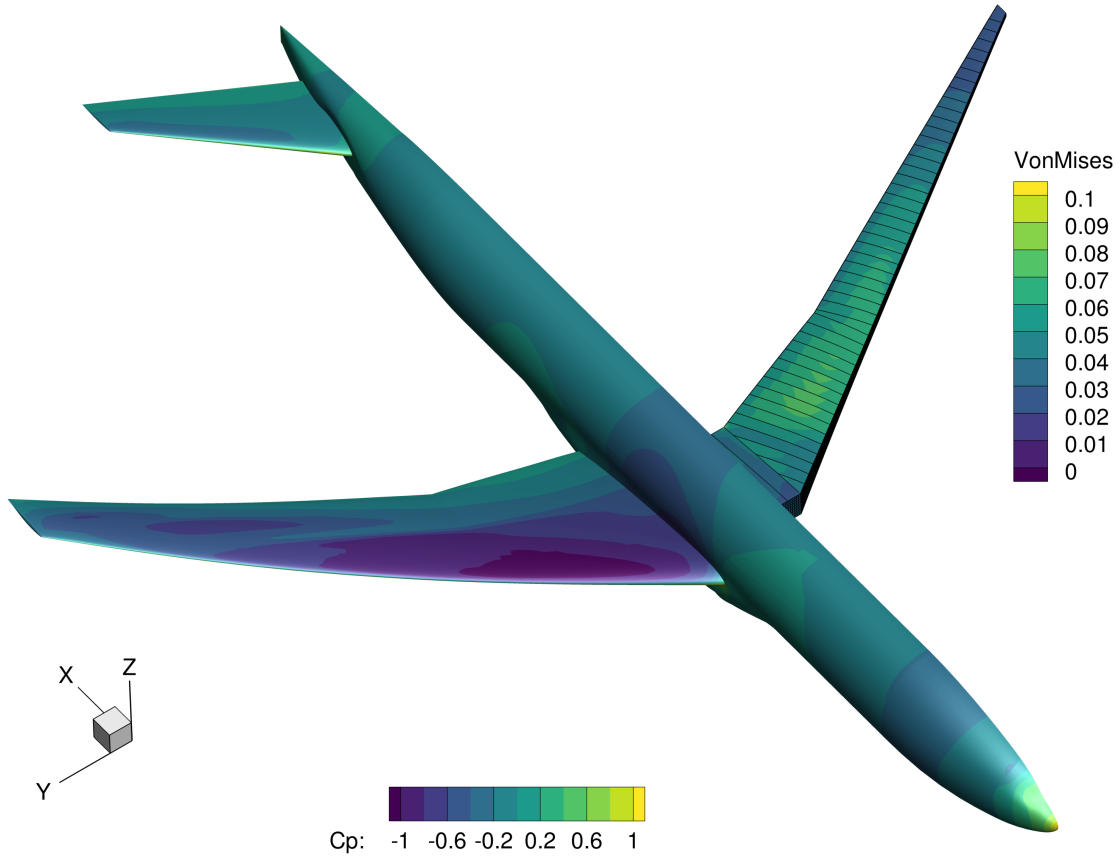


Figure 8: Common Research Model used for aerodynamic and aerostructural optimization examples

## 6.3 Aerodynamic optimization

### 6.3.1 Problem description

The aerodynamic optimization problem we solve is a two variable problem that can be stated as

$$\begin{aligned} &\text{Maximize : } ML/D \\ &\text{With respect to : } M, \alpha. \end{aligned} \tag{1}$$

This is a simple unconstrained optimization with a well-defined optimum and a smooth design space in the range of Mach and angles of attack of interest.

The script corresponding to this optimization is listed in Figure 9 and consists of less than 100 lines of code. There is no flow solver specific code in this script; it is possible to switch between flow solvers simply by changing the flow solver that is selected when creating the solver instance. Furthermore, the design of the API allows for a powerful optimization to be completed with a short and readable script, demonstrating the value of the work presented here.

Figure 9: Script that solves the aerodynamic optimization problem.

```
#Define the aerodynamic problem to solve.
ap = AeroProblem(name=apName, mach=Mach, reynolds=43e6, areaRef=areaRef, alpha=alpha,
                 chordRef=chordRef, reynoldsLength=chordRef, xRef=1325.90*.0254
                 zRef=177.95*.0254, T=298.15, evalFuncs=['l/d'])

# Add the aerodynamic design variables
ap.addDV('alpha', lower = 0.0, upper = alphaUpper, scale=numpy.pi/180.0)
ap.addDV('mach', lower = 0.5, upper = 0.89)
# Set the user options for the solver
aeroOptions = {user Options}
# Create the solver instance
CFDSolver = ADFLOW(options=aeroOptions)
# Define a function to compute the L/D
def LiftOverDrag(funcs):
    funcs['l/d'] = funcs['cl']/funcs['cd']
return funcs
CFDSolver.addUserFunction('l/d',['cl','cd'],LiftOverDrag)

# Define the function to evaluate the objective
def aeroFuncs(x):
    # Create the return dictionary
    funcs = {}
    # Set the design variables and evaluate the current Mach number
    ap.setDesignVars(x)
    ap.evalFunctions(funcs,['mach'])
    # Solve the flow and evaluate the functions of interest
    CFDSolver(ap)
    CFDSolver.evalFunctions(ap, funcs)
    CFDSolver.checkSolutionFailure(ap,funcs)
    # Compute the objective
    funcs['ml/d'] = -1.0*funcs['crm_mb_mach']*funcs['crm_mb_l/d']
    return funcs

def aeroFuncsSens(x, funcs):
    # Create the return dictionary
    funcsSens = {}
    # Evaluate the aeroProblem gradients
    ap.evalFunctionsSens(funcsSens,['mach'])
    # Evaluate the flow solver Gradients
    CFDSolver.evalFunctionsSens(ap, funcsSens)
    # Compute the composite gradient for the objective
    funcsSens['ml/d']={}
    for key in x:
        funcsSens['ml/d'][key] = -1.0*(funcs['crm_mb_l/d']*funcsSens['crm_mb_mach'][key]\
                                     +funcs['crm_mb_mach']*funcsSens['crm_mb_l/d'][key])
    return funcsSens

# Set-up Optimization Problem and Optimize
optProb = Optimization('opt', aeroFuncs, comm=MPI.COMM_WORLD)
# Add variables from the aeroProblem
ap.addVariablesPyOpt(optProb)
# Add Objective
optProb.addObj('ml/d', scale=1)
# Make Instance of Optimizer
optOptions = {user Options}
opt = OPT(args.optimizer, options=optOptions)
# Solve
sol = opt(optProb, aeroFuncsSens)
```

Because this type of aerodynamic shape optimization problem has a smooth design space, it is well-suited to gradient-based optimization [34, 35]. In this particular problem, it is



possible to visualize the design space because we have only two design variables, as shown in Figure 10. The contours show the variation of the  $ML/D$  objective over the region, while the lines show the optimization paths of the optimization from four different starting points. We use the algorithm developed by Kenway and Martins [42] to generate these contours.

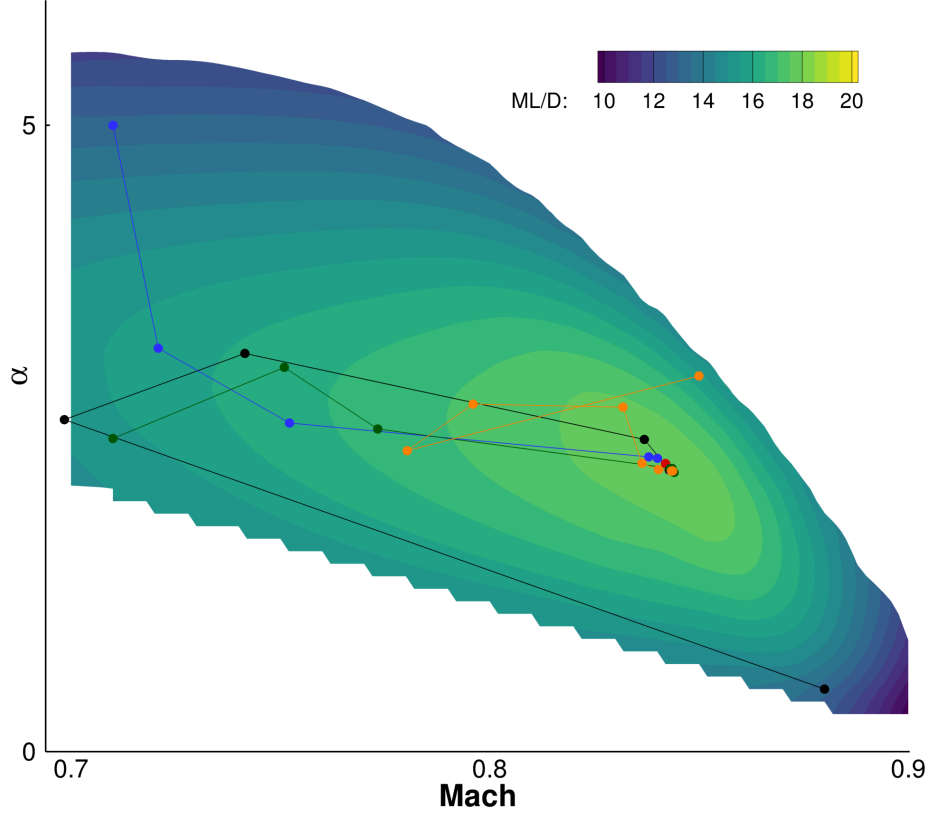


Figure 10: Aerodynamic  $ML/D$  optimization contours

The four starting points (listed in Table 5) were selected to provide four distinct combinations of  $\alpha$  and Mach number. These points are spread between high and low Mach numbers over a variety of angles of attack in the design space of interest. This shows the robustness of the optimization approach and provides a range of optimization results to use when analyzing the performance of the solver and API. While the four optimization starting points result in different convergence paths with different solution costs and number of flow evaluations, all four answers converge to the same optimum.

Table 5: Aerodynamic optimization starting points

	Point 1	Point 2	Point 3	Point 4
$\alpha$	0.5	5.0	2.5	3.0
$M$	0.88	0.71	0.71	0.85

Table 6: Aerodynamic optimization final points

	Point 1	Point 2	Point 3	Point 4
$\alpha$	2.24244	2.24273	2.24267	2.24270
$M$	0.84365	0.84364	0.84364	0.84364
$ML/D$	17.92385	17.92384	17.92386	17.92386

The results for the four different starting points are summarized in Tables 6. We show the values for the objective as well as both design variables. The objective and Mach number both match to the fifth decimal place for all optimizations, while the angle of attack varies in the fourth decimal place. This shows that all four cases have converged to the same point.

Table 7 shows the variation in computational cost across the four optimizations. The different starting points result in a range of  $\pm 20\%$  in the average solution cost relative to the overall average. The spread in the gradient cost is lower, at  $\pm 8\%$  of the overall average. The larger spread in the solution cost is largely due to the two failed solutions in run number two. Depending on the mode of failure, these failed solutions can take significant amounts of time to solve, driving up the overall average for that run.

Table 7: Convergence results for the four aerodynamic optimization starting points in TWU

	Point 1	Point 2	Point 3	Point 4	Average
Initialization cost	103.10	102.61	103.08	117.60	106.60
Average flow solution cost	7,448.88	9,497.69	8,209.91	7,252.73	8,102.30
Average function evaluation cost	0.12	0.12	0.24	0.24	0.18
Average gradient evaluation cost	7,804.28	7,602.13	6,957.70	7,935.68	7,574.95
Major iteration count	7	8	8	9	8
Total solution failures	0	2	0	0	0.5

To analyze the impact of our API and solver development recommendations on the above costs, we use the timings from the fourth point to look at a more detailed breakdown of the computational cost of each part of the solution process.

### 6.3.2 Initialization

The overall cost of the initialization in this example problem is 117.60 TWU, or 1.6% of the average solution time, which is negligible. A detailed breakdown of the cost for this call is shown in Table 8. The entries in this table reflect the fact that the initialization is only called once, i.e., the mean, minimum and maximum are the same and the standard deviation is zero. Since the initialization is only called once, that cost is amortized over the number of solutions required for the optimization.

The two largest costs in the initialization are the partitioning and the preprocessing. The partitioning includes the reading of the mesh file and splitting it to run in parallel. The

preprocessing time consists of the time required to set up the multi-grid structure, communication patterns, compute mesh metrics, and wall distances. Much of this computational effort is dependent only on the mesh size and topology, and does not need to be updated between flow solutions when using the library based solver approach. The exceptions to this are the mesh metric and wall distance computations, which need to be updated when the design geometry is changed for each optimization iteration. In the context of this fixed geometry optimization, this cost is small, but as we will see in the aerostructural analysis example, this cost can become prohibitive.

Table 8: Initialization cost breakdown (TWU)

	Mean	Median	Minimum	Maximum	Std. dev.
Library loading	1.00	1.00	1.00	1.00	0.00
Default variable setting	0.01	0.01	0.01	0.01	0.00
Base class initialization	0.04	0.04	0.04	0.04	0.00
Library initialization	0.41	0.41	0.41	0.41	0.00
Partitioning	21.40	21.40	21.40	21.40	0.00
Preprocessing	94.54	94.54	94.54	94.54	0.00
Family setup	0.00	0.00	0.00	0.00	0.00
Flow initialization	0.15	0.15	0.15	0.15	0.00
Total	117.60	117.60	117.60	117.60	0.00

### 6.3.3 Solution

Table 9 summarizes the variation of the flow solution computational cost over the course of a nominal optimization. The mean flow solution is over 98% of the total solution cost. If this were to hold for the entire optimization, the importance of the direct memory API would be diminished, since the cost of reading and writing the solution would be a small portion of the overall computational cost.

However, Figure 11 shows that there is a significant variation in solution time over the course of the optimization. Specifically, the solution times decrease significantly towards the end of the optimization. Looking at the minimum solution time case, the solution write time accounts for 12% of the solution time, which is much more significant. Furthermore, if we add in an additional 120 TWU for an initialization, a full 25% of the call time would be consumed by non-solution tasks. In such a case, the direct memory access API becomes much more significant, since 25% of the computational cost can be avoided by not re-initializing the solver and writing out a solution for every function evaluation.

The full savings from the direct memory API can be assessed by comparing the cost of re-initializing the solver to the cost of setting the AeroProblem. Setting the AeroProblem updates the flow conditions for the upcoming flow solution, getting the solver ready to analyze the new point without fully re-initializing the solver. Table 9 shows that this costs about 0.4 TWU on average, which is far less expensive than a full initialization.

Table 9: Solution cost breakdown (TWU)

	Mean	Median	Minimum	Maximum	Std. dev.
Set AeroProblem	0.39	0.35	0.08	1.60	0.28
Flow solution	7,162.41	6,921.13	564.39	23,261.70	4,808.93
Write solution	89.93	85.76	75.73	126.12	12.67
Total	7,252.73	7,013.91	643.69	23,342.30	4,808.09

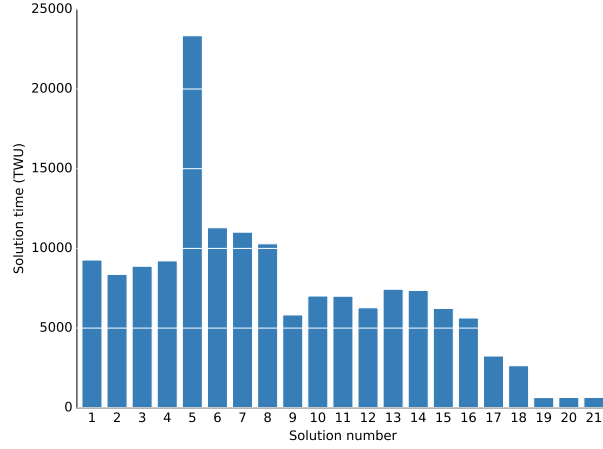


Figure 11: Evolution of flow solution cost during optimization.

### 6.3.4 Function analysis

The function analysis is the process of computing the engineering values of interest from the converged states of the CFD solver. These are quantities such as  $C_L$  and  $C_D$  that are needed as outputs of the analysis. This also includes the computation of any user defined functions, such as  $L/D$ .

Table 10 shows the function evaluation costs in work units. These computations are negligible relative to the cost of solving the flow.

Table 10: Function evaluation time (TWU)

	Mean	Median	Minimum	Maximum	Std. dev.
Set AeroProblem	0.15	0.10	0.10	0.49	0.12
Function evaluation	0.09	0.03	0.03	0.59	0.17
User function evaluation	0.00	0.00	0.00	0.00	0.00
Total function evaluation	0.24	0.12	0.12	1.01	0.29

### 6.3.5 Gradient computation

In ADflow, we use the adjoint method to compute the gradients of the flow solution. This method requires the solution of a large linear system to compute the adjoint vector, and then a matrix-vector product to compute the total derivative of the desired function with respect to the design variables. In this case, we only have two design variables, but in a typical case we would have hundreds of variables, making the adjoint approach an efficient method for computing derivatives [11].

Table 11, lists the costs of computing the gradients. These costs are split between the cost of computing the adjoint solution and the cost computing the total derivative for the given adjoint vector. The adjoint solution is the dominant part of this cost, but with a single output function and multiple design variables, it is the most efficient way to compute gradients.

The total gradient computation cost is similar to that of the flow solution. The major difference is that the standard deviation in computational cost is lower than for the flow solution. This is because the adjoint system is only solved if the flow solution is successful; therefore, the adjoint equations are always more likely to converge and there are fewer failed adjoint solutions in a typical optimization, if any.

The other contribution to the smaller standard deviation is that restarting the adjoint from a previous solution does not benefit the adjoint solution as much as for the flow solution. In most cases, the adjoint is only evaluated after a change in state values, so it is likely that the adjoint equations have changed between evaluations. However, since the flow solver is accessible as a library through memory, the cost of starting with a previous adjoint solution is negligible, so it is standard practice to do this.

Table 11: Evaluation costs for gradient of  $L/D$  with respect to cruise Mach and angle-of-attack (TWU).

	Mean	Median	Minimum	Maximum	Std. dev.
Adjoint solution	7,924.01	8,010.54	5,419.97	10,747.64	1,417.50
Total derivative equation	11.47	11.47	11.45	11.52	0.02
Total	7,935.68	8,022.12	5,431.54	10,759.19	1,417.59

## 6.4 Aerostructural analysis

To consider the characteristics of a multidisciplinary problem, we start by considering a simple aerostructural analysis, which is again based on the CRM geometry shown in Figure 8. The flow condition is the starting point for the Point 1 optimization case. Once again, we analyze the relative costs of the various preprocessing, initialization, and solution tasks. In this case, we report metrics for the aerodynamic and structural solver, as well as the load and displacement transfer object, and the overall aerostructural solution process.

### 6.4.1 Initialization

The cost of initializing the aerostructural problem includes the cost of initializing the aerodynamic and structural solvers, as well as the cost of linking the two solvers with a load and displacement transfer object. The costs of these initializations are listed in Table 12. The cost of initializing the aerostructural class itself is insignificant, so it is not shown here. The cost of initializing the CFD solver is the same as before—approximately 100 TWU, while the structural initialization is faster—only 20 TWU. The single largest cost in the initialization is the setup of the load and displacement transfer object, which takes almost 1000 TWU. This is due to the search algorithm that finds the connections between the aerodynamic surface and the structural mesh [5].

On its own, the cost of the initialization is not that significant. Given the approach outlined in this paper, this cost is only incurred a single time and accounts for less than 5% of the total solution cost. However, as we discuss in Section 6.4.2, if the initialization cost is incurred repeatedly for aerodynamic and structural solutions in the multidisciplinary solution process, the overhead associated with it quickly becomes significant.

Table 12: Initialization cost of aerostructural analysis (TWU)

	Mean	Median	Minimum	Maximum	Std. dev.
Aerodynamic solver	97.11	97.11	97.11	97.11	0.00
Structural solver	20.33	20.33	20.33	20.33	0.00
Transfer object	976.43	976.43	976.43	976.43	0.00
Total	1,093.87	1,093.87	1,093.87	1,093.87	0.00

### 6.4.2 Aerostructural Solution

The total solution cost for the aerostructural equations is about three times the solution cost of the aerodynamic solution alone and requires 30 aerostructural iterations to converge five orders of magnitude. In this case, an aerostructural iteration is a single Gauss–Seidel iteration between the aerodynamic solver and the structural solver, where the aerodynamic solver is the first to run.

We typically perform partial solutions—one or two orders of magnitude reduction of the residual norm—for each solver in each aerostructural iteration to limit wasted computation.

The computational cost of a single aerostructural solution is shown in Table 13. The majority of this computational cost is due to the CFD solver, which takes over 95% of the cost. The average structural solution cost is roughly 100 times faster than the CFD solution, even when including the stiffness matrix factorization required in the first structural solution. However, the low cost of all of the ancillary computations is largely made possible by the methods outlined in this work.

If file I/O were used to couple the aerodynamic and structural solvers, the best case scenario would require reading and writing a file with the force and displacement transfer for

every aerostructural iteration. In that scenario, we are comparing the cost of those operations to the cost of a single aerostructural iteration, rather than the entire aerostructural process. If we count just the cost of the aerodynamic initialization (97 TWU from Table 12) and writing the aerodynamic and structural solutions (145 TWU from Table 13, we would get a cost of 242 TWU. This compares to 760 TWU for an average aerostructural iteration, which is just over 30% of additional cost. In the more likely scenario of a simple file I/O wrapper that treats the three components as independent executables, we would need to re-do all three initializations, the structural matrix factorization, and the solution writing for each aerostructural iteration. In that scenario, the average single iteration cost would increase from 760 TWU to 2186 TWU—almost three times the current cost. This makes it clear that it is imperative to develop solvers as libraries, have them loaded in memory, and pass data directly, to avoid replicating unnecessary work and driving up the cost of the multidisciplinary analysis and optimization.

Table 13: Aerostructural solution cost (TWU)

	Mean	Median	Minimum	Maximum	Std. dev.
Structural solution total	7.83	2.00	1.98	188.31	32.42
Aerodynamic solution total	753.11	750.47	72.56	1,655.40	416.92
Aerostructural problem setup	12.43	12.43	12.43	12.43	–
Load and displacement transfer	0.06	0.06	0.06	0.06	–
Structural pre-solution	188.33	188.33	188.33	188.33	–
Gauss–Seidel solution	23,426.62	23,426.62	23,426.62	23,426.62	–
Solution writing	145.08	145.08	145.08	145.08	–
Aerostructural solution total	23,772.54	23,772.54	23,772.54	23,772.54	–

### 6.4.3 Coupled gradient computation

To perform aerostructural optimization, we require the coupled gradients, which are computed using a coupled-adjoint approach [5]. The aerostructural gradient computation cost, shown in Table 14, is roughly 13% more expensive than the equivalent aerodynamic adjoint computation. This is a much smaller increase in cost than the full nonlinear system. The main source of this efficiency comes from the coupled-Krylov solution algorithm used to solve the linear system, which is more efficient than the simpler Gauss–Seidel algorithm used to solve the nonlinear aerostructural problem itself. A full description of these methods and their application to the solution of both the aerostructural system and the corresponding coupled adjoint system is presented by Kenway et al. [5]. The cost of the aircraft mass gradient is significantly smaller than the  $L/D$  adjoint because the mass is independent of both the aerodynamic and structural states, so the adjoint vector is zero and there is no requirement to solve the coupled linear system. In this table, the statistical values are constant because there is only one linear solution computed for a single analysis, since a fully-coupled Krylov approach is used to solve the coupled linear system.

The implementation of the coupled-Krylov algorithm for the coupled adjoint linear system is only possible because we treat all disciplines in the multidisciplinary systems as libraries that provide direct memory access for all components. This allows a fully-coupled linear system to be setup using the gradient computation methods in each discipline, which can only be done efficiently through direct memory access. This also simplifies the computation of the coupled derivatives in the off-diagonal elements of the coupled Jacobian. Because both solvers are available in memory, the coupled chain-rule accumulation of the derivatives needed for the coupled adjoint is easier to set up. This accumulation would still be possible using file I/O, but it would be much less efficient because the amount of data that would need to be written to and read from disk would be high relative to the problem size.

Table 14: Aerostructural gradient computation cost (TWU)

	Mean	Median	Minimum	Maximum	Std. dev.
Adjoint setup	0.27	0.27	0.27	0.27	–
Adjoint for $L/D$	8,651.79	8,651.79	8,651.79	8,651.79	–
Gradient for mass	191.43	191.43	191.43	191.43	–
Total derivative equation	58.21	58.21	58.21	58.21	–
Total	8,901.68	8,901.68	8,901.68	8,901.68	–

## 6.5 Aerostructural optimization

The final demonstration is an aerostructural optimization, where the optimization problem formulation is the same as before:

$$\begin{aligned} \text{Maximize : } & ML/D \\ \text{With respect to : } & M, \alpha. \end{aligned} \tag{2}$$

However, the wing flexibility is now accounted for because the objective function evaluations are based on aerostructural computations. This shifts the optimal result because the flexibility of the wing alters the value and intercept of the lift curve slope for the aircraft. As in the aerostructural analysis above, the coupling with the structural discipline adds several more components in the timing analysis, which are summarized in Table 15.

The breakdown of the cost for the full optimization history is similar to the values breakdown for the single point analysis, but there are some minor differences. The average total aerostructural solution cost is lower than for the single-point case. This is mostly because as the optimization progress, the cost of individual solutions tends to decrease, reducing the overall average. This is also the reason why the average number of iterations (flow solutions) per aerostructural solution averages only 16 over the full set of optimizations, rather than 30 for the single point case. There are also flow solution failures in optimizations 2, 3, and 4, which reinforces the points made in Section 2 regarding the graceful handling of solution failures.



Table 15: Breakdown for the cost (in TWU) for aerostructural (AS) optimizations starting from four different points.

	Point 1	Point 2	Point 3	Point 4	Average
Flow initialization	113.69	115.81	116.21	114.90	115.15
Structural initialization	20.89	29.29	20.39	20.41	22.75
Transfer initialization	978.81	985.34	975.31	986.62	981.52
Average flow solution	1,048.44	1,043.49	1,035.46	928.12	1,013.88
Average structural solution	2.97	2.74	2.59	2.39	2.67
Average AS solution	18,726.47	21,579.10	14,589.84	11,276.55	16,542.99
Average AS function evaluation	0.51	0.49	0.47	0.47	0.49
Average AS gradient computation	7,922.07	7,262.83	7,450.85	7,743.60	7,594.84
Major iteration count	8	9	7	8	8.00
Total solution failures	0	2	1	1	1.00
Total flow solutions	284	595	378	590	461.75
Total structural solutions	300	624	405	639	492.00
Total AS solutions	16	29	27	49	30.25
Average flow solutions per AS solution	18	21	14	12	16

## 7 Conclusion

The requirements for a flow solver used in multidisciplinary analysis and optimization are different than the requirements for a stand-alone flow solver. We introduce a series of requirements that are necessary for an efficient multidisciplinary solver, discuss the idea of treating the flow solver as a library (rather than a stand-alone code), and introduce an API that makes it possible to set up complex multidisciplinary analysis and optimization problems using compact scripts written in a high-level language.

As a specific example of a flow solver following these guidelines, we introduce the open-source CFD solver ADflow.<sup>6</sup> Using ADflow, we quantify the impact of these requirements on the computational performance of aerodynamic optimization, aerostructural analyses, and aerostructural optimization.

We show that for aerodynamic optimization with a direct memory access API saves 12% to 25% of the optimization time, while for an aerostructural optimization, the cost can be reduced by up to a factor of three compared to a file I/O based approach. These results conclusively demonstrate the benefits of having a direct memory access API for multidisciplinary analysis codes.

ADflow has already been used extensively to investigate aerodynamic and aerostructural design optimization problems. ADflow is part of a wider aerodynamic shape optimization tool suite (MACH-Aero), which is also available under an open-source license. Several of those investigations resulted in open benchmarks that can be built on by other researchers. In addition, the ADflow API could be re-used for other flow solvers, which could then be used interchangeably in the MACH-Aero framework.

<sup>6</sup><https://github.com/mdolab/adflow>, accessed March 2020

## 8 Acknowledgments

Resources supporting this work were provided by the NASA High-End Computing (HEC) Program through the NASA Advanced Supercomputing (NAS) Division at Ames Research Center.

## Appendix A: A brief history of ADflow

The core solver in ADflow is derived from Sumb [74], which is a structured, multi-block CFD solver with a second-order finite-volume discretization, that was developed in the early 2000's at Stanford University. In addition to the basic second order finite-volume scheme, Sumb has the capability to run steady, unsteady, and time-spectral cases with fixed or moving meshes, and it has several turbulence models available, including Spalart–Allmaras,  $k$ - $\omega$ ,  $k$ - $\omega$  SST,  $k$ - $\tau$ , and v2-f. The mean flow is solved with an explicit Runge–Kutta multi-grid scheme and the turbulence models are all solved with a segregated DDADI scheme. This code was originally developed to run as a stand alone executable, reading options from a text input file. The code was written in module-based Fortran 90 with MPI for parallelization and was demonstrated to successfully solve large CFD problems using thousands of cores [74].

ADflow has been under development since 2006 in the Multidisciplinary Design Optimization Laboratory. Several significant features have been added over this period of development. These include an adjoint solver for steady-state Euler equations in 2006–2007 [75] and a time-spectral solver in 2012 [76], as well as a discrete adjoint for the RANS equations in 2013 [77]. In 2015, the RANS adjoint solver was updated with a Jacobian-free adjoint solver by Kenway et al. [11], greatly reducing the memory requirements of the adjoint solver. The Python API presented in this work was developed from 2006 to 2013 and has remained largely unchanged since the end of that period. The addition of the adjoint solver facilitated the implementation of the NK solver, which significantly improved the performance of the code for optimization. This original NK solver implementation has recently been superseded by the implementation of a more robust approximate ANK algorithm by Yildirim et al. [58]. The ability to solve overset meshes was also added in 2016–2017 by Kenway et al. [71].

ADflow was released as open-source software on March 28, 2019, under the GNU Lesser General Public License (LGPL), version 2.1.

## References

- [1] Venkatamaran, S., and Haftka, R. T., “Structural optimization complexity: what has Moore’s law done for us?” *Structural and Multidisciplinary Optimization*, Vol. 28, 2004, pp. 375–387. doi:10.1007/s00158-004-0415-y.
- [2] Parkinson, C., “Parkinson’s Law or the pursuit of progress.” *The Economist*, 1959.
- [3] Thimbleby, H., “Viewpoint. Computerised Parkinson’s law,” *Computing & Control Engineering Journal*, Vol. 4, No. 5, 1993, pp. 197–198. doi:10.1049/cce:19930049.

- [4] Vassberg, J. C., DeHaan, M. A., Rivers, M. S., and Wahls, R. A., “Retrospective on the Common Research Model for Computational Fluid Dynamics Validation Studies,” *Journal of Aircraft*, Vol. 55, No. 4, 2018, pp. 1325–1337. doi:10.2514/1.C034906.
- [5] Kenway, G. K. W., Kennedy, G. J., and Martins, J. R. R. A., “Scalable Parallel Approach for High-Fidelity Steady-State Aeroelastic Analysis and Derivative Computations,” *AIAA Journal*, Vol. 52, No. 5, 2014, pp. 935–951. doi:10.2514/1.J052255.
- [6] Kenway, G. K. W., “A Scalable, Parallel Approach for Multi-Point, High-Fidelity Aerostructural Optimization of Aircraft Configurations,” Ph.D. thesis, University of Toronto, 2013.
- [7] Yu, Y., Lyu, Z., Xu, Z., and Martins, J. R. R. A., “On the Influence of Optimization Algorithm and Starting Design on Wing Aerodynamic Shape Optimization,” *Aerospace Science and Technology*, Vol. 75, 2018, pp. 183–199. doi:10.1016/j.ast.2018.01.016.
- [8] Jameson, A., “Aerodynamic Design via Control Theory,” *Journal of Scientific Computing*, Vol. 3, No. 3, 1988, pp. 233–260. doi:10.1007/BF01061285.
- [9] Giles, M. B., and Pierce, N. A., “An Introduction to the Adjoint Approach to Design,” *Flow, Turbulence and Combustion*, Vol. 65, 2000, pp. 393–415. doi:10.1023/A:1011430410075.
- [10] Martins, J. R. R. A., and Hwang, J. T., “Review and Unification of Methods for Computing Derivatives of Multidisciplinary Computational Models,” *AIAA Journal*, Vol. 51, No. 11, 2013, pp. 2582–2599. doi:10.2514/1.J052184.
- [11] Kenway, G. K. W., Mader, C. A., He, P., and Martins, J. R. R. A., “Effective Adjoint Approaches for Computational Fluid Dynamics,” *Progress in Aerospace Sciences*, Vol. 110, 2019, p. 100542. doi:10.1016/j.paerosci.2019.05.002.
- [12] He, P., Mader, C. A., Martins, J. R. R. A., and Maki, K. J., “An Aerodynamic Design Optimization Framework Using a Discrete Adjoint Approach with OpenFOAM,” *Computers & Fluids*, Vol. 168, 2018, pp. 285–303. doi:10.1016/j.compfluid.2018.04.012.
- [13] Gazaix, M., Jollès, A., and Lazareff, M., “The elsA object-oriented computational tool for industrial applications,” *Proceeding of the ICAS 2002 Congress*, ICAS, 2002.
- [14] Cambier, L., Heib, S., and Plot, S., “The ONERA elsA CFD software: input from research and feedback from industry,” *Mechanics and Industry*, Vol. 14, No. 3, 2013. doi:10.1051/meca/2013056.
- [15] Weller, H. G., Tabor, G., Jasak, H., and Fureby, C., “A tensorial approach to computational continuum mechanics using object-oriented techniques,” *Computers in Physics*, Vol. 12, No. 6, 1998, pp. 620–631. doi:10.1063/1.168744.

- [16] Jasak, H., Jemcov, A., and Tuković, Z., “OpenFOAM: A C++ Library for Complex Physics Simulations,” *International Workshop on Coupled Methods in Numerical Dynamics, IUC*, Citeseer, 2007.
- [17] Kennedy, G. J., and Martins, J. R. R. A., “A parallel aerostructural optimization framework for aircraft design studies,” *Structural and Multidisciplinary Optimization*, Vol. 50, No. 6, 2014, pp. 1079–1101. doi:10.1007/s00158-014-1108-9.
- [18] Drela, M., “XFOIL: An Analysis and Design System for Low Reynolds Number Airfoils,” *Low Reynolds Number Aerodynamics*, edited by T. J. Mueller, Springer Berlin Heidelberg, Berlin, Heidelberg, 1989, pp. 1–12. doi:10.1007/978-3-642-84010-4\_1.
- [19] Gray, J. S., Hwang, J. T., Martins, J. R. R. A., Moore, K. T., and Naylor, B. A., “OpenMDAO: An open-source framework for multidisciplinary design, analysis, and optimization,” *Structural and Multidisciplinary Optimization*, Vol. 59, No. 4, 2019, pp. 1075–1104. doi:10.1007/s00158-019-02211-z.
- [20] Hwang, J. T., and Martins, J. R. R. A., “A computational architecture for coupling heterogeneous numerical models and computing coupled derivatives,” *ACM Transactions on Mathematical Software*, Vol. 44, No. 4, 2018, p. Article 37. doi:10.1145/3182393.
- [21] Yildirim, A., Kenway, G. K. W., Mader, C. A., and Martins, J. R. R. A., “A Jacobian-free approximate Newton–Krylov startup strategy for RANS simulations,” *Journal of Computational Physics*, Vol. 397, 2019, p. 108741. doi:10.1016/j.jcp.2019.06.018.
- [22] Kenway, G. K., Kennedy, G. J., and Martins, J. R. R. A., “A CAD-Free Approach to High-Fidelity Aerostructural Optimization,” *Proceedings of the 13th AIAA/ISSMO Multidisciplinary Analysis Optimization Conference*, Fort Worth, TX, 2010. doi:10.2514/6.2010-9231.
- [23] Jameson, A., Schmidt, W., and Turkel, E., “Numerical Solution of the Euler Equations by Finite Volume Methods Using Runge–Kutta Time Stepping Schemes,” *14th Fluid and Plasma Dynamics Conference*, 1981. doi:10.2514/6.1981-1259.
- [24] Turkel, E., and Vatsa, V. N., “Effects of Artificial Viscosity on Three-Dimensional Flow Solutions,” *AIAA Journal*, Vol. 32, 1994, pp. 39–45. doi:10.2514/3.11948.
- [25] van Leer, B., “Towards the ultimate conservative difference scheme. V. A second-order sequel to Godunov’s method,” *Journal of Computational Physics*, Vol. 32, 1979, pp. 101–136. doi:10.1016/0021-9991(79)90145-1.
- [26] Roe, P. L., “Approximate Riemann Solvers, Parameter Vectors, and Difference Schemes,” *Journal of Computational Physics*, Vol. 43, 1981, pp. 357–372. doi:10.1016/0021-9991(81)90128-5.
- [27] Spalart, P., and Allmaras, S., “A One-Equation Turbulence Model for Aerodynamic Flows,” *La Recherche Aérospatiale*, Vol. 1, 1994, pp. 5–21.

- [28] Wilcox, D. C., *Turbulence Modeling for CFD*, 3<sup>rd</sup> ed., DCW Industries, Inc., La C nada, CA, 2006.
- [29] Menter, F. R., “Two-equation eddy-viscosity turbulence models for engineering applications,” *AIAA Journal*, Vol. 32, No. 8, 1994, pp. 1598–1605. doi:10.2514/3.12149.
- [30] Rumsey, C., “NASA Turbulence Modeling Resource,” <https://turbmodels.larc.nasa.gov>, 2019. Accessed: 2019-03-27.
- [31] Klopfer, G., Hung, C., Van der Wijngaart, R., and Onufer, J., “A diagonalized diagonal dominant alternating direction implicit (D3ADI) scheme and subiteration correction,” *29th AIAA, Fluid Dynamics Conference, Albuquerque, NM*, 1998. doi:10.2514/6.1998-2824.
- [32] Balay, S., Gropp, W. D., McInnes, L. C., and Smith, B. F., “Efficient Management of Parallelism in Object Oriented Numerical Software Libraries,” *Modern Software Tools for Scientific Computing*, edited by E. Arge, A. M. Bruaset, and H. P. Langtangen, Birkh user Press, 1997, pp. 163–202. doi:10.1007/978-1-4612-1986-6\_8.
- [33] Li, J., Bouhlel, M. A., and Martins, J. R. R. A., “Data-based Approach for Fast Airfoil Analysis and Optimization,” *AIAA Journal*, Vol. 57, No. 2, 2019, pp. 581–596. doi:10.2514/1.J057129.
- [34] He, X., Li, J., Mader, C. A., Yildirim, A., and Martins, J. R. R. A., “Robust aerodynamic shape optimization—from a circle to an airfoil,” *Aerospace Science and Technology*, Vol. 87, 2019, pp. 48–61. doi:10.1016/j.ast.2019.01.051.
- [35] Lyu, Z., Kenway, G. K. W., and Martins, J. R. R. A., “Aerodynamic Shape Optimization Investigations of the Common Research Model Wing Benchmark,” *AIAA Journal*, Vol. 53, No. 4, 2015, pp. 968–985. doi:10.2514/1.J053318.
- [36] Lyu, Z., and Martins, J. R. R. A., “Aerodynamic Shape Optimization of an Adaptive Morphing Trailing Edge Wing,” *Journal of Aircraft*, Vol. 52, No. 6, 2015, pp. 1951–1970. doi:10.2514/1.C033116.
- [37] Kenway, G. K. W., and Martins, J. R. R. A., “Multipoint Aerodynamic Shape Optimization Investigations of the Common Research Model Wing,” *AIAA Journal*, Vol. 54, No. 1, 2016, pp. 113–128. doi:10.2514/1.J054154.
- [38] Chen, S., Lyu, Z., Kenway, G. K. W., and Martins, J. R. R. A., “Aerodynamic Shape Optimization of the Common Research Model Wing-Body-Tail Configuration,” *Journal of Aircraft*, Vol. 53, No. 1, 2016, pp. 276–293. doi:10.2514/1.C033328.
- [39] Mader, C. A., and Martins, J. R. R. A., “Stability-Constrained Aerodynamic Shape Optimization of Flying Wings,” *Journal of Aircraft*, Vol. 50, No. 5, 2013, pp. 1431–1449. doi:10.2514/1.C031956.

- [40] Lyu, Z., and Martins, J. R. R. A., “Aerodynamic Design Optimization Studies of a Blended-Wing-Body Aircraft,” *Journal of Aircraft*, Vol. 51, No. 5, 2014, pp. 1604–1617. doi:10.2514/1.C032491.
- [41] Secco, N. R., and Martins, J. R. R. A., “RANS-based Aerodynamic Shape Optimization of a Strut-braced Wing with Overset Meshes,” *Journal of Aircraft*, Vol. 56, No. 1, 2019, pp. 217–227. doi:10.2514/1.C034934.
- [42] Kenway, G. K. W., and Martins, J. R. R. A., “Buffet Onset Constraint Formulation for Aerodynamic Shape Optimization,” *AIAA Journal*, Vol. 55, No. 6, 2017, pp. 1930–1947. doi:10.2514/1.J055172.
- [43] Mangano, M., and Martins, J. R. R. A., “Multipoint Aerodynamic Shape Optimization for Subsonic and Supersonic Regimes,” *57th AIAA Aerospace Sciences Meeting, AIAA SciTech Forum, 2019*, San Diego, CA, 2019. doi:10.2514/6.2019-0696.
- [44] Brelje, B. J., and Martins, J. R. R. A., “Coupled component sizing and aerodynamic shape optimization via geometric constraints,” *AIAA AVIATION Forum*, American Institute of Aeronautics and Astronautics, Dallas, TX, 2019. doi:10.2514/6.2019-3105.
- [45] Brelje, B. J., Anibal, J., Yildirim, A., Mader, C. A., and Martins, J. R., “Flexible Formulation of Spatial Integration Constraints in Aerodynamic Shape Optimization,” *AIAA Journal*, 2020. (In press).
- [46] Hwang, J. T., Jasa, J., and Martins, J. R. R. A., “High-fidelity design-allocation optimization of a commercial aircraft maximizing airline profit,” *Journal of Aircraft*, Vol. 56, No. 3, 2019, pp. 1165–1178. doi:10.2514/1.C035082.
- [47] Kenway, G. K. W., and Martins, J. R. R. A., “Multipoint High-Fidelity Aerostructural Optimization of a Transport Aircraft Configuration,” *Journal of Aircraft*, Vol. 51, No. 1, 2014, pp. 144–160. doi:10.2514/1.C032150.
- [48] Liem, R. P., Kenway, G. K. W., and Martins, J. R. R. A., “Multimission Aircraft Fuel Burn Minimization via Multipoint Aerostructural Optimization,” *AIAA Journal*, Vol. 53, No. 1, 2015, pp. 104–122. doi:10.2514/1.J052940.
- [49] Brooks, T. R., Kenway, G. K. W., and Martins, J. R. R. A., “Benchmark Aerostructural Models for the Study of Transonic Aircraft Wings,” *AIAA Journal*, Vol. 56, No. 7, 2018, pp. 2840–2855. doi:10.2514/1.J056603.
- [50] Brooks, T. R., and Martins, J. R. R. A., “On Manufacturing Constraints for Tow-steered Composite Design Optimization,” *Composite Structures*, Vol. 204, 2018, pp. 548–559. doi:10.1016/j.compstruct.2018.07.100.
- [51] Brooks, T. R., Martins, J. R. R. A., and Kennedy, G. J., “High-fidelity Aerostructural Optimization of Tow-steered Composite Wings,” *Journal of Fluids and Structures*, Vol. 88, 2019, pp. 122–147. doi:10.1016/j.jfluidstructs.2019.04.005.

- [52] Burdette, D. A., and Martins, J. R. R. A., “Design of a Transonic Wing with an Adaptive Morphing Trailing Edge via Aerostructural Optimization,” *Aerospace Science and Technology*, Vol. 81, 2018, pp. 192–203. doi:10.1016/j.ast.2018.08.004.
- [53] Burdette, D. A., and Martins, J. R. R. A., “Impact of Morphing Trailing Edge on Mission Performance for the Common Research Model,” *Journal of Aircraft*, Vol. 56, No. 1, 2019, pp. 369–384. doi:10.2514/1.C034967.
- [54] He, S., Jonsson, E., Mader, C. A., and Martins, J. R. R. A., “A Coupled Newton–Krylov Time-Spectral Solver for Wing Flutter and LCO Prediction,” *AIAA Aviation Forum*, Dallas, TX, 2019. doi:10.2514/6.2019-3549.
- [55] He, S., Jonsson, E., Mader, C. A., and Martins, J. R. R. A., “Aerodynamic Shape Optimization with Time Spectral Flutter Adjoint,” *2019 AIAA/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference*, American Institute of Aeronautics and Astronautics, San Diego, CA, 2019. doi:10.2514/6.2019-0697.
- [56] He, S., Jonsson, E., Mader, C. A., and Martins, J. R. R. A., “A Coupled Newton–Krylov Time Spectral Solver for Flutter Prediction,” *2018 AIAA/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference*, American Institute of Aeronautics and Astronautics, Kissimmee, FL, 2018. doi:10.2514/6.2018-2149.
- [57] Gray, J. S., Mader, C. A., Kenway, G. K. W., and Martins, J. R. R. A., “Modeling Boundary Layer Ingestion Using a Coupled Aeropropulsive Analysis,” *Journal of Aircraft*, Vol. 55, No. 3, 2018, pp. 1191–1199. doi:10.2514/1.C034601.
- [58] Yildirim, A., Gray, J. S., Mader, C. A., and Martins, J. R. R. A., “Aeropropulsive Design Optimization of a Boundary Layer Ingestion System,” *AIAA Aviation Forum*, Dallas, TX, 2019. doi:10.2514/6.2019-3455.
- [59] Gray, J. S., and Martins, J. R. R. A., “Coupled Aeropropulsive Design Optimization of a Boundary-Layer Ingestion Propulsor,” *The Aeronautical Journal*, Vol. 123, No. 1259, 2019, pp. 121–137. doi:10.1017/aer.2018.120.
- [60] Gray, J. S., Kenway, G. K. W., Mader, C. A., and Martins, J. R. R. A., “Aero-propulsive Design Optimization of a Turboelectric Boundary Layer Ingestion Propulsion System,” *2018 AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference*, Atlanta, GA, 2018. doi:10.2514/6.2018-3976, AIAA 2018-3976.
- [61] Kenway, G. K., and Kiris, C. C., “Aerodynamic Shape Optimization of the STARC-ABL Concept for Minimal Inlet Distortion,” *AIAA/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference*, American Institute of Aeronautics and Astronautics, 2018. doi:10.2514/6.2018-1912.
- [62] Madsen, M. H. A., Zahle, F., Sørensen, N. N., and Martins, J. R. R. A., “Multipoint high-fidelity CFD-based aerodynamic shape optimization of a 10 MW wind turbine,” *Wind Energy Science*, Vol. 4, 2019, pp. 163–192. doi:10.5194/wes-4-163-2019.

- [63] Dhert, T., Ashuri, T., and Martins, J. R. R. A., “Aerodynamic Shape Optimization of Wind Turbine Blades Using a Reynolds-Averaged Navier–Stokes Model and an Adjoint Method,” *Wind Energy*, Vol. 20, No. 5, 2017, pp. 909–926. doi:10.1002/we.2070.
- [64] Garg, N., Kenway, G. K. W., Lyu, Z., Martins, J. R. R. A., and Young, Y. L., “High-fidelity Hydrodynamic Shape Optimization of a 3-D Hydrofoil,” *Journal of Ship Research*, Vol. 59, No. 4, 2015, pp. 209–226. doi:10.5957/JOSR.59.4.150046.
- [65] Garg, N., Pearce, B. W., Brandner, P. A., Phillips, A. W., Martins, J. R. R. A., and Young, Y. L., “Experimental Investigation of a Hydrofoil Designed via Hydrostructural Optimization,” *Journal of Fluids and Structures*, Vol. 84, 2019, pp. 243–262. doi:10.1016/j.jfluidstructs.2018.10.010.
- [66] Garg, N., Kenway, G. K. W., Martins, J. R. R. A., and Young, Y. L., “High-fidelity Multipoint Hydrostructural Optimization of a 3-D Hydrofoil,” *Journal of Fluids and Structures*, Vol. 71, 2017, pp. 15–39. doi:10.1016/j.jfluidstructs.2017.02.001.
- [67] Liao, Y., Garg, N., Martins, J. R. R. A., and Young, Y. L., “Viscous Fluid Structure Interaction Response of Composite Hydrofoils,” *Composite Structures*, Vol. 212, 2019, pp. 571–585. doi:10.1016/j.compstruct.2019.01.043.
- [68] Vassberg, J. C., DeHaan, M. A., Rivers, S. M., and Wahls, R. A., “Development of a Common Research Model for Applied CFD Validation Studies,” 2008. doi:10.2514/6.2008-6919.
- [69] Kennedy, G. J., and Martins, J. R. R. A., “A Parallel Finite-Element Framework for Large-Scale Gradient-Based Design Optimization of High-Performance Structures,” *Finite Elements in Analysis and Design*, Vol. 87, 2014, pp. 56–73. doi:10.1016/j.finel.2014.04.011.
- [70] Mader, C. A., Kenway, G. K., Martins, J. R. R. A., and Uranga, A., “Aerostructural Optimization of the D8 Wing with Varying Cruise Mach Numbers,” *18th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference*, American Institute of Aeronautics and Astronautics, 2017. doi:10.2514/6.2017-4436.
- [71] Kenway, G. K. W., Secco, N., Martins, J. R. R. A., Mishra, A., and Duraisamy, K., “An Efficient Parallel Overset Method for Aerodynamic Shape Optimization,” *Proceedings of the 58th AIAA/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference, AIAA SciTech Forum*, Grapevine, TX, 2017. doi:10.2514/6.2017-0357.
- [72] Secco, N. R., Jasa, J. P., Kenway, G. K. W., and Martins, J. R. R. A., “Component-based Geometry Manipulation for Aerodynamic Shape Optimization with Overset Meshes,” *AIAA Journal*, Vol. 56, No. 9, 2018, pp. 3667–3679. doi:10.2514/1.J056550.



- [73] Gray, J. S., Chin, J., Hearn, T., Hendricks, E., Lavelle, T., and Martins, J. R. R. A., “Chemical Equilibrium Analysis with Adjoint Derivatives for Propulsion Cycle Analysis,” *Journal of Propulsion and Power*, Vol. 33, No. 5, 2017, pp. 1041–1052. doi:10.2514/1.B36215.
- [74] van der Weide, E., Kalitzin, G., Schluter, J., and Alonso, J. J., “Unsteady Turbomachinery Computations Using Massively Parallel Platforms,” *Proceedings of the 44th AIAA Aerospace Sciences Meeting and Exhibit*, Reno, NV, 2006. doi:10.2514/6.2006-421, AIAA 2006-0421.
- [75] Mader, C. A., Martins, J. R. R. A., Alonso, J. J., and van der Weide, E., “ADjoint: An Approach for the Rapid Development of Discrete Adjoint Solvers,” *AIAA Journal*, Vol. 46, No. 4, 2008, pp. 863–873. doi:10.2514/1.29123.
- [76] Mader, C. A., and Martins, J. R. R. A., “Derivatives for Time-Spectral Computational Fluid Dynamics Using an Automatic Differentiation Adjoint,” *AIAA Journal*, Vol. 50, No. 12, 2012, pp. 2809–2819. doi:10.2514/1.J051658.
- [77] Lyu, Z., Kenway, G. K., Paige, C., and Martins, J. R. R. A., “Automatic Differentiation Adjoint of the Reynolds-Averaged Navier–Stokes Equations with a Turbulence Model,” *21st AIAA Computational Fluid Dynamics Conference*, San Diego, CA, 2013. doi:10.2514/6.2013-2581.