AD flow is a free and open-source computational fluid dynamics solver for aerodynamic and interdisciplinary optimization.

Charles A. Mader, Gaetan K. W. Kenway, Anil Yildirim,
University of Michigan, Ann Arbor, Michigan, 48109

Abstract

It is now normal practice to simulate fluid behavior numerically by solving the Navier-Stokes equations using turbulence models. To efficiently optimize a design using a flow solver in the loop, however, it is necessary to do more than just solve these equations. An efficient aerodynamic and interdisciplinary design optimizer is discussed, along with the guidelines for constructing such a flow solver. The ability to load the flow solver as a library that gives you immediate memory access to the necessary data is one of the most highly recommended features. Some further suggestions include using a higher-level language for scripting and paying close attention to solution warm beginning, code efficiency, flow solver robustness, and solution failure handling. We provide the open-source flow solver ADflow as an example of a tool that adheres to these guidelines. The performance benefits suggested by the proposals are supported by the outcomes of aerodynamic optimization, aerostructural analysis, and aerostructural optimization performed using ADflow. The release of these suggestions and the availability of the source code opens the door for other solvers to use the same application programming interface. ADflow is an open-source component of a larger set of tools for optimizing aerodynamic shapes.

1 Introduction

Because the availability and power of computers has improved, computational approaches have been applied differently in engineering design. Formerly reserved for final design verifications, computationally expensive simulations are now routinely used throughout the preliminary design phase. Many uses may be found for this newfound potential: One, simulations may be run at a finer spatial or temporal grain. Complex physical models may be used in simulations; for instance, the Reynolds-Averaged Navier-Stokes (RANS) equations can be solved in place of the simpler Euler equations.

This allows for a wider range of flight configurations to be examined within a given geometry.

Four, parameter sweeps or an optimization algorithm may be used to compare and contrast many designs in search of improvement and a better knowledge of the design performance tradeoffs.

5. A multidisciplinary study may be carried out by combining computational models that are typically studied separately.

Venkatamaran and Haftka [1] studied the historical implications of growing computing performance on structural analysis and optimization. They observed that time spent on computational analysis tends to grow to occupy all available time, in accordance with Parkinson's Law [2]. A similar rule by Thimbleby [3] asserts that software programs develop to take up greater computer memory, processing capability, and storage space. Venkata- maran and Haftka [1] also note out that anecdotal evidence shows that time needed for “adequate” structural analysis has remained

consistent, at 6 to 8 hours over the previous 30 years. Situations 1 and 2 above are exemplified by the use of refined computational models made possible by computing advancements. We think that the key reason for this is because the first two situations alone do not substantially impact the complexity of an engineering design work flow.

In this study, we analyze the criteria required to accomplish scenarios 3 through 5 in the context of computational fluid dynamics (CFD) (CFD). In all three of these cases, the computational approach must be employed frequently by fully automated means, necessitating the inclusion of extra elements essential to accurate calculation. One of these must-have characteristics is a clear and simple API that allows the solver to be used as a built library with direct memory access (API). This enables the solver to be deployed successfully as part of an analytical framework on large-scale high-performance computing (HPC) resources.

In this work, we show how ADflow, a free, open-source, structured, multi-block, overset flow solver, satisfies these needs.

1 For the CRM shape specifically, aerodynamic and aerostructural design optimization challenges are solved by ADflow [4, 5]. All of the calculations described in this study are steady-state RANS solutions. A specific fidelity option or approach to solving the problem is not necessary to understand or implement the ideas presented here. The API has been used to 2D and 3D panel solvers as well as various 3D flow solvers. Apart than the shown steady-state solutions, ADflow is also capable of time-accurate and time-spectral computations.

What follows is a summary of the paper. In Part 2, we go into depth about what a multi-disciplinary solver needs to do, and in Section 3, we present the idea of the solver as a library of code. In Section 4, we detail the Python API that was created to meet the requirements laid forth in Chapter 2. These sections are supposed to provide a general guidance for solver needs and are thus solver agnostic. Part 5 discusses how these conditions were accomplished for the ADflow solver, and includes an overview of past investigations made feasible by ADflow. Many analysis and optimizations are presented in Section 6 to show how well ADflow performs. Section 7 outlines the key results of this investigation.

2 Requirements for an efficient multidisciplinary flow solver

Unlike a standalone solver, a flow solver designed for interdisciplinary analysis or optimization requires a particular set of features. In order to solve for a certain geometry and flow condition with enough engineering precision, a standalone flow solver must be as fast as feasible. To solve a multidisciplinary analysis or optimization issue, the flow solver works as part of a bigger framework and has to be performed numerous times in succession without operator intervention. As you'll see later, this has various ramifications for the necessary features of the flow solver.

2.1 Solution failure handling

2.2

When employing a CFD solver inside an automated process, as is essential for interdisciplinary analysis and optimization, the solver is typically needed to examine a large range of operating points without human interaction. In this case, it is possible that the flow solver will be instructed to execute one or more analyses that fail to yield an acceptable solution. For a completely automated method, the solution must fail gracefully without incurring an unrecoverable defect. This is especially essential for HPC simulations since the total process

might takes many hours or days, and aborted operations incur the extra expense of resubmitting the task and waiting for its turn in the queue.

2.3 As this is the case, the solution for an automated process must include robust exception handling. Whether the solution completely converges, diverges, partly converges, stalls, or generates a NaN (not a number), these are all significant events that must be handled. Keeping an eye on the residual of the flow equations and using logic trees makes these scenarios quite straightforward to deal with. After processing each scenario, a boolean response may be returned to the user or the driving algorithm.

2.4 The primary exception to this is the circumstance when NaNs are encountered. In this instance, the solver has to be reset, including a thorough re-initialization of the flow to guarantee that all of the NaNs in memory are purged so that future flow solutions are not cancelled because of a prior NaN in memory. Although while it is more expensive to reset the flow solution entirely, this is still far more efficient than re-initializing the flow solver and reallocating all the necessary memory as a standalone solver would need to do.

2.5 Solution restart

2.6 The need to automatically execute several solutions in succession implies, secondly, that there is a strong incentive to reduce the cost of each solution in the series. A solution restart technique is the quickest and easiest way to do this, since it ensures that subsequent solutions begin in a convergent condition. For many optimization tasks, like generating drag polars, sweeping parameters, and conducting gradient-based optimizations, the previously-solved state is preferable to the default uniform flow. This can be done using file I/O for most solutions, but doing it in memory is significantly quicker. Combining solution restarts with the right method may significantly increase the pace at which subsequent solutions are found. For example, when the solver is restarted using the prior answer as the beginning estimate, Newton's approach produces great terminal convergence given a good starting point. This restart feature should be made available as an option via the API, enabling the user to deactivate it if beginning with a uniform flow is more advantageous, in circumstances when the prior solution is not an appropriate starting point.

2.7 Robust startup

No matter the application, a reliable approach for initializing the flow solver is always welcome. Yet, as we compute a series of successive answers in an automated form, it is crucial that our launch procedure be as bulletproof as possible. The automated process will generate a large set of initial conditions, from which the solver must converge to a solution.

Moreover, this is necessary if the flow is reset because of a poor solution, necessitating a fresh start at the following solution point. On top of that, while optimizing a design, the optimizer is likely to test out impractical intermediate designs, such as those with extreme flow separation. For these special instances, Newton's approach often fails during the first phases of convergence, while producing strong terminal convergence. The necessity for a solid beginning procedure is

heightened by these elements.

2.8 Fine-grained iteration control

When utilizing a flow solver in a coupled analysis (for example, coupled with an external structural solver or a propulsion model) it is also vital to be able to manage the number of iterations conducted for a particular solution. In many of these circumstances, it has been proven that completing partial flow solutions between coupling updates enables the coupled solution to be completed with just a minimal increase in the overall flow solution cost. This has been proved for example, in static aeroelastic analysis by Kenway et al. [5], who showed that a relative convergence tolerance of 0.1 per Efficient convergence through all phases of solution

Three distinct stages can be identified in the external aerodynamic simulations we calculate using ADflow: initialization, steady state, and shutdown. The near-field of the aerodynamic surface interacts with the initial flow solution during the start-up phase. The near-field solution and the far-field boundary interact, and the flow solver deals with it at the transition analysis stage. Finally, the solver converges the numerical solution of the flow to further minimize the residuals to the set convergence tolerance, after having successfully captured the overall flow patterns.

The first two stages are often the main focus of traditional engineering flow models. For this reason, it is sufficient to converge through the transition phase of the solution only far enough to instill engineering confidence in the solution. All three stages of the solution become crucial, however, since throughout an optimization, tight numerical convergence of the solution is desired, particularly towards the conclusion of the optimization process.

Each of these stages is characterized by a distinct kind of convergence for various algorithms. This highlights the need for a flexible solution algorithm switching mechanism that may be used throughout each simulation to improve convergence rates. The nonlinear residual norm provides a useful gauge for tracking progress toward convergence and should trigger these transitions automatically.

2.9 Direct memory access and API

Most optimization and interdisciplinary analyses use file I/O to interact with standalone flow solvers. With this method, a guiding framework or script is set up to mechanically produce solver and optimizer input and output files. Although while this method may be used to pair codes practically anywhere, it does have certain downsides.

The first problem is that there isn't enough disk bandwidth, which increases the time it takes to store, retrieve, and process the data. As file I/O is often a shared resource on massively parallel computing platforms, the throughput experienced by a particular user might be significantly reduced. If you have a powerful network connecting your computing nodes, you can exchange data at rates that are exponentially higher than anything you could ever hope to achieve with file I/O. Yet, information transmission becomes much more challenging when numerous disciplines or optimizers make use of simultaneous solution methodologies.

The possibility for inaccurate results is the second downside of the file I/O method. When data is reloaded, some information may be lost in comparison to an original double precision reference if the analysis output is written using ASCII with a restricted number of digits or using binary with single precision (to conserve disk space and I/O time). In order to get rid of this disparity, binary double precision numbers must be used for all stored data, which results in a lot of space being used up on the disk.

The third problem is that the same standalone code is always being run. A new process must be started whenever a code is called, and one-off initialization operations are often carried out at the outset. This part of the code is

usually not as speed-optimized as the rest of the program. Moreover, in the course of optimization and cross-disciplinary analysis, iterative solutions are used.

tions have a lot in common with one another. If you're using an iterative approach, you may save time and money by carrying over this data from solution to solution. While a restart capability is not strictly necessary for this process, it does increase the data that must be written to and retrieved from the disk. As a result of these two variables, it is far more costly to do many subsequent analyses with a standalone code than with the same code executed as a library with an API.

If an application programming interface (API) does not provide direct access to the necessary functions, as is often the case with commercial software, then the file I/O strategy is the only remaining alternative. In order to prevent problems with file I/O, it is essential that all data transfer from the CFD code be performed through direct memory access. The analysis code is built as a library instead of a standalone executable, and a process script controls the order in which tasks are executed during optimization or analysis. Each succeeding analysis is set up to be executed directly by the process script rather than through an input file. This makes it possible for the aerodynamic states, forces, and gradients, among other data, to be transported across memory on their way into and out of the CFD solver. As a result, the cost each iteration drops dramatically, since there is no longer any need to write data to disk. When we are updating coupling variables or switching iteration techniques, there is no penalty to halting and beginning the iteration process as we move variables via memory. For a static aeroelastic solution using ADflow, Kenway [6] analyzes the costs of the direct memory access and file I/O techniques and finds that the I/O approach is twice as expensive.

2.10 Code efficiency

Multidisciplinary analysis and optimization codes place a higher premium on code efficiency. The repeated code runs used to iteratively refine the design are mostly responsible for this. It's already costly to run analytic procedures, and this iterative process just makes it more so. So, it is crucial to enhance the computational efficiency of analytical codes used in interdisciplinary design optimization.

Specifically, we take into account these three tiers of efficiency when we design our code. Algorithmic efficiency is the first and most crucial level, and it is reached by using cutting-edge algorithms to converge the linear and nonlinear systems of equations that emerge throughout the optimization process. The next step in efficiency is a direct-memory-access API, which, as was previously indicated, removes the effect of file I/O constraints on the solver's speed. Code optimizations tailored to the running algorithms and hardware are necessary for maximum efficiency. Memory bandwidth constraints may be reduced, vectorization can be optimized, and the number of cache misses can be reduced, among other related enhancements.

2.11 Additional requirements for efficient multidisciplinary design opti-mization

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 multi- disciplinary 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 ap- proaches 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 functionalityto 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 whenthe 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 un- derlying 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 sweepover a range of angle of attack variables. Figure 2 details the

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 runonly 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 unreal-istic 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 Aero-Solver 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.

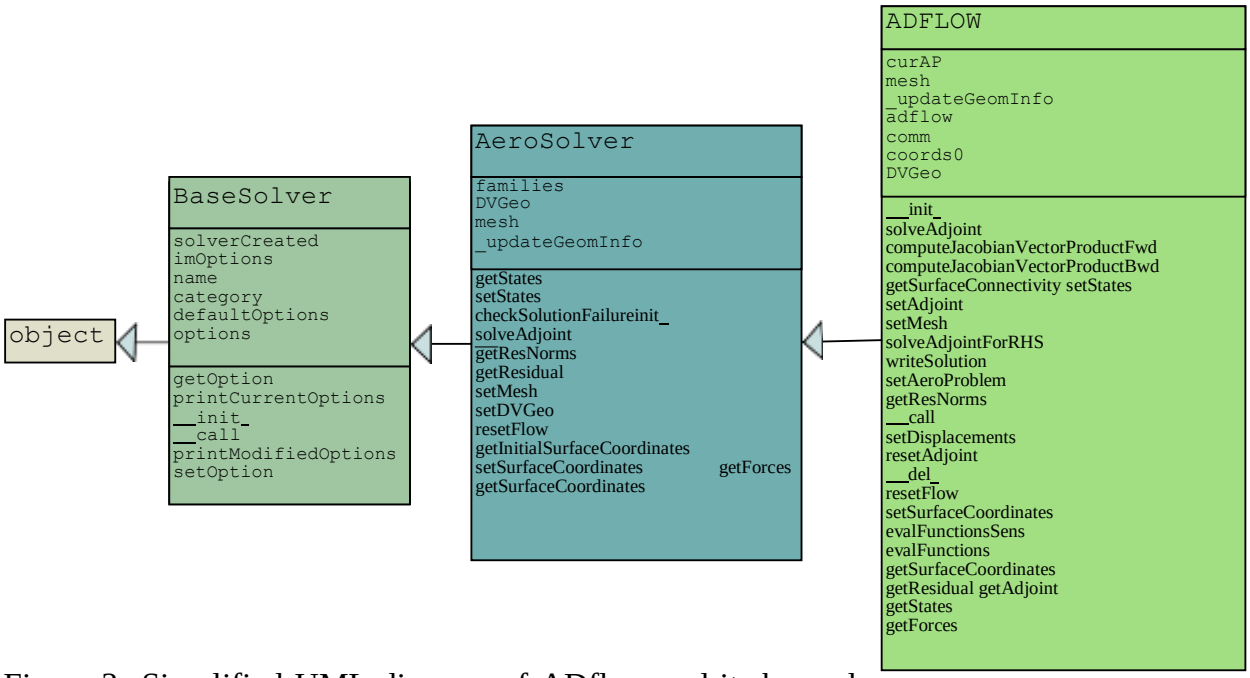

Figure 3: Simplified UML diagram of ADflow and its base classes.

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.

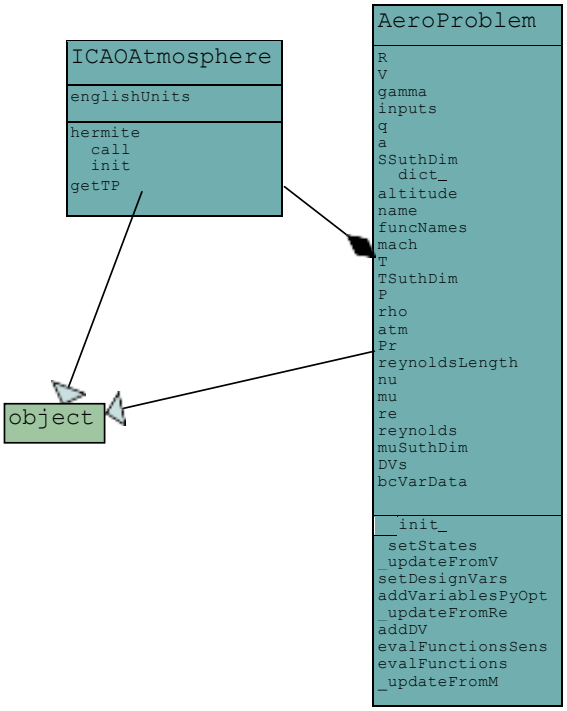

Figure 4: Simplified UML of the aerodynamic problem class.

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.

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 previously mentioned, thephilosophy of

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 aero-dynamic 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 ofthe 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 co- ordinates. 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 inputand 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.

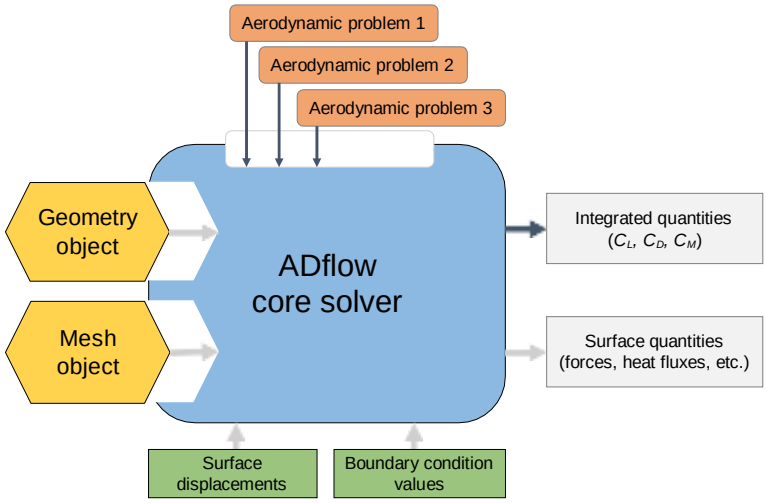

Figure 7: ADflow integration with other components and disciplines in a multidisciplinary context.

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 4th 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 solu- tion. 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 ap- proach 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 differenti- ation 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 men- tioned 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 require- ments, 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 de- velopers 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 de- sired 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 underly-ing 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 Conclusion

When it comes to interdisciplinary analysis and optimization, a flow solver's needs are distinct from those of a standalone flow solver. We present a list of needs for a powerful multidisciplinary solver, discuss the concept of using the flow solver as a library (rather than a standalone piece of code), and present an application programming interface (API) that allows for the configuration of intricate multidisciplinary analysis and optimization problems through the use of concise scripts written in a high-level language.

We offer the free and open-source CFD solver ADflow as an example of a flow solver that adheres to these standards. We use ADflow to measure the effect of these constraints on the speed with which aerodynamic optimization, aerostructural analysis, and aerostructural optimization may be performed computationally.

We demonstrate that a direct memory access API saves 12% to 25% of the optimization time for aerodynamic optimization, and that it may lower the cost of aerostructural optimization by as much as a factor of three when compared to a file I/O based method. These findings prove without a reasonable doubt how useful a direct memory access API would be for scripts doing transdisciplinary analyses.

Aerodynamic and aerostructural design optimization issues have previously been explored using ADflow. ADflow is included in a more comprehensive set of open-source software for optimizing aerodynamic shapes (MACH-Aero). Some of these studies have produced publicly available standards for further study. Also, new flow solvers might be developed using the ADflow API and utilized interchangeably inside the MACH-Aero framework.

References

S. Venkatamaran and R. T. Haftka, "Structural optimization complexity: what has Moore's law done for us?" Optimization in Structural and Multidisciplinary Contexts, Volume 28, Issue 3, Pages 375-387, 2004. doi:10.1007/s00158-004-0415-y.

According to "Parkinson's Law or the Pursuit of Progress" [2] by C. Parkinson. Reference: The Economist (1959).

Perspective," in H. Thimbleby (ed. A Computerized Version of Parkinson's Law," Computers & Control Engineering, Volume 4, Issue 5, 1993, Pages 197–198. doi:10.1049/cce:19930049.

Reference: [4]J. C. Vassberg, M. A. DeHaan, M. S. Rivers, and R. A. Wahls, "Retrospective on the Common Research Model for Computational Fluid Dynamics Validation Studies," Journal of Aviation, Vol. 55, No. 4, 2018, pp. 1325-1337. doi:10.2514/1.C034906.

Scalable Parallel Approach for High-Fidelity Steady-State Aeroelastic Analysis and Derivative Computations. Kenway, G. K. W., Kennedy, G. J., and Martins, J. R. R. A. AIAA Journal, Volume 52, Issue 5 (May 2014), Pages 935–951.

Using a Scalable, Parallel Method to Multi-Point, High-Fidelity Aerostructural Optimization of Aircraft Configurations. (2013). Kenway, G. K. W., Ph.D. Thesis, University of Toronto.

"On the Impact of Optimization Algorithm and Beginning Design on Wing Aerodynamic Shape Optimization" by Yu, Y., Lyu, Z., Xu, Z., and Martins, J. R. R. A., published in Aerospace Science and Technology, Volume 75, 2018, pages 183-199, doi:10.1016/j.ast.2018.01.016.

Journal of Scientific Computing, vol. 3, no. 3, 1988, pp. 233-260, Jameson, A., "Aerodynamic Design using Control Theory," doi:10.1007/BF01061285.

An Introduction to the Adjoint Method to Design, by M. B. Giles and N. A. Pierce, Flow, Turbulence, and Combustion, Volume 65, 2000, Pages 393-415. doi:10.1023/A:1011430410075.

[10] Martins, J. R. R. A., and Hwang, J. T., "Review and Unification of Techniques for Com- puting Derivatives of Multidisciplinary Computational Models," AIAA Journal, Volume 51, No 11, 2013, Pages 2582-2599. doi:10.2514/1.J052184.

Progress in Aerospace Sciences, Volume 110, Issue 100542, March 2019, pages G. K. W. Kenway, C. A. Mader, P. He, and J. R. R. A. Martins, "Effective Adjoint Methods for Computational Fluid Dynamics," DOI:10.1016/j.paerosci.2019.05.002.

An Aerodynamic Design Optimization Framework Utilizing a Discrete Adjoint Method with OpenFOAM, by He, P., Mader, C. A., Martins, J. R. R. A., and Maki, K. J., Computers & Fluids, Vol. 168, 2018, pp. 285-303. doi:10.1016/j.compfluid.2018.04.012.

[13] Gazaix, M., Joll`es, A., and Lazareff, M., "The elsA object-oriented computational tool for industrial applications," Proceeding of the ICAS 2002 Congress, ICAS, 2002.

[14]

The ONERA elsA CFD software: input from research and feedback from industry," by L. Cambier, S. Heib, and S. Plot; Mechanics and Industry; Volume 14; Issue 3; 2013; doi:I10.1051/meca/2013056.

[15]

Weller, H. G., Tabor, G., Jasak, H., and Fureby, C., "A tensorial approach to computa- tional continuum mechanics using object-oriented techniques," Computers in Physics, Vol. 12, No. 6, 1998, pp. 620–631. doi:10.1063/1.168744.

OpenFOAM is a C++ library for complex physics simulations [16] Jasak, H., A. Jemcov, and Z. Tukovi'c, International Symposium on Coupled Techniques in Numerical Dynamics, IUC, Citeseer, 2007.

[17] Kennedy, G. J., and Martins, J. R. R. A., "A parallel aerostructural optimization frame- work 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 Air- foils," 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.

Reference: Gray, J. S., J. T. Hwang, J. R. R. A. Martins, K. T. Moore, and B. A. Naylor, "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 et al., "A computational architecture for linking het- erogeneous numerical models and calculating coupled derivatives," ACM Transactions on Mathematical Software, vol. 44, no. 4, article 37, 2018. doi:10.1145/3182393.

[21] A. Yildirim, G. K. W. Kenway, C. A. Mader, and J. R. R. A. Martins, "A Jacobian- free approximation Newton-Krylov starting approach for RANS simulations," Journal of Computational Physics, vol. 397, no. 10, 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 Method 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.

14th Fluid and Plasma Dynamics Conference, 1981, p. 59, Jameson, A., Schmidt, W., and Turkel, E. "Numerical Solution of the Euler Equations by Finite Volume Techniques Utilizing Runge-Kutta Time Stepping Schemes." 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.

With reference to [25] van Leer, B., "Towards the ultimate conservative difference scheme. Journal of Computational Physics, Vol. 32, 1979, pp. 101-136, doi:10.1016/0021-9991(79)90145-1, Part V. A Second-Order Sequel to Godunov's Method.

"Approximate Riemann Solvers, Parameter Vectors, and Difference Schemes," by P. L. Roe, was published in 1981 in Journal of Computational Physics, Volume 43, pages 357–372.

[27]

A One-Equation Turbulence Model for Aerodynamic Flows, by Pedro Spalart and Salvador Allmaras, was published in La Recherche Aerospatiale, Volume 1, Issue 1, 1994, Pages 5–21.

Turbulence Modeling for Computational Fluid Dynamics, Third Edition, Wilcox, David C., DCW Industries, Inc., La Caada, CA, 2006.

Reference: [29] Menter, F. R., "Two-equation eddy-viscosity turbulence models for engineering applications," AIAA Journal, Volume 32, Issue 8 (1994), Pages 1598-1605. doi:10.2514/3.12149.

[30]

A NASA Turbulence Modeling Library, by C. Rumsey, https://turbmodels.larc.nasa.gov, 2019. Accessed: 2019-03-27.