Loose Coupling of Isolated Rotor-blade Rotorcraft CFD/CSD Simulations using preCICE

submitted in fulfillment of the requirements for the degree of Master of Science (M.Sc.) to the Faculty of Informatics of the Technical University of Munich.

Supervising Professors	Univ.-Prof. Dr. Hans-Joachim Bungartz Chair of Scientific Computing in Computer Science
	Prof. Dr. Ing. Manfred Hajek Institute of Helicopter Technology
Supervised by	Abdelmoula, Amine M.Sc.
	Chourdakis, Gerasimos M.Sc.
Submitted by	Huang Qunsheng
Registration number	03693984
Document ID	HT-MA 193/2019
Submitted on	Garching, December 23, 2019

Statement of Authorship

I, Huang Qunsheng, confirm that the work presented in this thesis has been performed and interpreted solely by myself except where explicitly identified to the contrary. All verbatim extracts have been distinguished by quotation marks, and all sources of information have been specifically acknowledged. I confirm that this work has not been submitted elsewhere in any other form for the fulfillment of any other degree or qualification.

Garching, 23 December 2019

Abstract

Accurate rotor blade analysis is a complex task—requiring the consideration of the aerodynamics, motion and deformation of the rotor blade. The current paradigm is to employ two separate solvers, each adept at solving for a single physical phenomenon in the multi-physics problem. In this work, the fluid solver TAU and the solid solver CAMRAD II are coupled. TAU is highly specialized in solving fluid problems around complex geometries, while CAMRAD II describes elastic blade behavior and provides trimmed solutions. A rudimentary fluid-solid solver coupling between TAU and CAMRAD II has been developed at the Chair of Helicopter Technologies [1]. However, the coupling is only intended for a single simulation case. As a result, the code is not designed to be modular, extensible or easy to use. Additionally, if this methodology would be used for future couplings, a unique adapter would be required for each simulation case.

Hence, this work focuses on the development of *general* adapters that support the "plug-and-play" of new solvers using the preCICE library. The structure of the adapter code is designed to be modular, using an Object-Oriented approach. The adapters minimize code dependencies between the two solvers and allow for the modular removal or addition of specific pre- or post-processing steps when passing data between solvers. Additionally, the TAU adapter allows the design and introduction of new simulation loops using the `TAU-Python` API.

Additional methods were implemented to supplement the preCICE library, which allowed for the passing of multiple timesteps of data in a single coupling step in a loosely-coupled simulation. Similarly, an intermediate interpolation step was implemented to allow for the 3D-1D passing of data from TAU to CAMRAD II.

Three cases were implemented using the adapter code. A simple, tightly-coupled simulation was used in the development of the TAU adapter as a proof of concept. In this case, the TAU adapter replaced the `OpenFOAM` adapter in an existing `OpenFOAM-CalculiX` coupling. The second case was a loosely-coupled rotor blade simulation with only rigid body motion, where we saw preliminary

convergence of TAU and CAMRAD II results. This simulation was extended to introduce elastic deformation in addition to rigid body motion to create a third case. The aim was to model elastic blade motion in the fluid simulation.

This work acts as a foundation for the TAU and CAMRAD II preCICE adapters, serving as a modular, extensible and easy to maintain codebase upon which other TAU or CAMRAD II fluid-structure interaction simulations can be developed. The adapter code also allows users to freely couple these two solvers with any other existing preCICE adapters and provides a standard interface to couple with new solvers in the future.

Acknowledgements

Firstly, I am grateful to Amine Abdelmoula and Gerasimos Chourdakis, my two supervisors, for guiding me through my work on this thesis. I would not have been able to accomplish a fraction of the work done in this thesis without their constant willingness to put their own work on hold to provide much-needed guidance and advice. I must thank Amine for his constant cheer and enthusiasm and Gerasimos for his unending calm that kept me centered throughout this work.

Additionally, I must thank Dominik Komp and Stefan Platzer at the Chair for Helicopter Technologies for their technical guidance in using the CAMRAD II and TAU solvers. Similarly, I must thank Isaan Desai for sharing his experiences on running preCICE on the lrz supercomputers.

Finally, I must thank my friends and family for their undending support for the duration of my studies. Special thanks to my brother, who always keeps me grounded, and my parents, without whom none of this would be possible.

Contents

List of Figures **xii**

List of Tables **xv**

List of Algorithms **xv**

Nomenclature **xvii**

1 Introduction **1**
- 1.1 Motivation 1
- 1.2 Structure of Work 2

2 Literature Review **5**

3 Theoretical Principles **7**
- 3.1 Monolithic vs. Partitioned Coupling Strategies 7
- 3.2 Tightly-Coupled vs. Loosely-Coupled Rotor Blade Simulations . 8
- 3.3 Introduction to Chimera Grids 8
- 3.4 Introduction to Radial Basis Functions 9

4 Tools **11**
- 4.1 TAU 11
  - 4.1.1 TAU Simulations 11
  - 4.1.2 TAU-Python API 12
  - 4.1.3 TAU Data Input 14
  - 4.1.4 TAU Data Output 16
- 4.2 CAMRAD II 17
  - 4.2.1 CAMRAD II Simulations 17
  - 4.2.2 CAMRAD II Data Input 19
  - 4.2.3 Force Tables 19
  - 4.2.4 CAMRAD II Data Output 20
- 4.3 TAU & CAMRAD II Coordinate Systems 20
- 4.4 preCICE 22
  - 4.4.1 Data Mapping 23
  - 4.4.2 preCICE Coupling 24
  - 4.4.3 preCICE API 27

5 Adapter Architecture **31**
5.1 Goals . . . 31
5.2 Workflow of Original Code . . . 32
5.3 Workflow with TAU and CAMRAD II Adapters . . . 35
5.4 TAU Adapter API . . . 43
5.4.1 Adapter class . . . 43
5.4.2 DataInterface class . . . 44
5.4.3 Config class . . . 45
5.4.4 DataHandler class . . . 47
5.4.5 Solver class . . . 49
5.5 CAMRAD II Adapter API . . . 51
5.5.1 Adapter class . . . 51
5.5.2 DataInterface class . . . 52
5.5.3 DataHandler class . . . 53
5.5.4 Config class . . . 54

6 Implementation **55**
6.1 Toy Example: Perpendicular Flap . . . 55
6.1.1 Simulation Setup . . . 56
6.1.2 TAU Solver . . . 57
6.1.3 TAU DataHandlers . . . 57
6.1.4 Folder Structure & Running Tutorial Case . . . 57
6.2 Case 1: Loose Coupling without Deformation . . . 59
6.2.1 Simulation Setup . . . 59
6.2.2 Simulation Coupling Workflow . . . 62
6.2.3 CAMRAD II Solver . . . 63
6.2.4 TAU Solver . . . 64
6.2.5 Data Handling . . . 65
6.2.6 Folder Structure & Running Tutorial Case . . . 74
6.3 Case 2: Loose Coupling with Deformation . . . 76
6.3.1 Simulation Setup . . . 76
6.3.2 Deformation Calculation . . . 76
6.3.3 Folder Structure & Running Tutorial Case . . . 79

7 Results **81**
7.1 Toy Example: Perpendicular Flap . . . 81
7.2 Case 1: Loose Coupling without Deformation . . . 82
7.3 Profiling . . . 85
7.4 Case 2: Loose Coupling with Deformation . . . 88

8 Conclusion **93**

References **97**

Appendices **103**

Appendix A User Guide **105**
A.1 Getting Started with TAU 105
A.1.1 Level 1: Using pre-existing Solver and DataHandlers . . 105
A.1.2 Level 2: Creating new DataHandlers and Config classes . 108
A.1.3 Level 3: Creating a new Solver class 109
A.2 Getting Started with CAMRAD II 109
A.2.1 Loosely-Coupled Helicopter Simulation 110

Appendix B Configuration File Definitions **113**
B.1 preCICE Configuration File 113
B.2 Parameters 113
B.3 TAU Configuration File 113
B.4 CAMRAD Configuration File 116
B.5 Parameters 116

Appendix C User Extensibility **121**
C.1 Extending TAU Adapter 121
C.1.1 Updating Config class 121
C.1.2 Extending DataHandler 123
C.1.3 Extending Solver class 126
C.2 Extending CAMRAD II Adapter 129
C.2.1 Updating Config class 129
C.2.2 Update DataHandler class 130

Appendix D Recreation of Data **133**
D.1 Toy Example: Perpendicular Flap 133
D.2 Case 1: Loose Coupling without Deformation 133
D.2.1 Coupling Results 133
D.2.2 Profiling Results 133
D.3 Case 2: Loose Coupling with Deformation 134

List of Figures

3.1 Chimera Grid Example showing Chimera Block and Cross-section. 9

4.1 TAU Parafile Excerpt. 12
4.2 TAU Motion Hierarchy Excerpt. 14
4.3 TAU Motion Hierarchy Excerpt. 15
4.4 TAU Scatfile Excerpt. 16
4.5 CAMRAD II Jobfile Structure 18
4.6 CAMRAD II Force Table Excerpt. 19
4.7 Coordinate Definitions used in TAU 21
4.8 Rotor Blade Reference Conventions. 21
4.9 Consistent Mapping of Temperatures 23
4.10 Conservative Mapping of Forces 23

5.1 Original Workflow Sequential UML Diagram. 33
5.2 Original Coupling Folder Structure. 34
5.3 Overview of TAU and CAMRAD II Adapters. 35
5.4 Updated TAU UML Diagram. 36
5.5 Updated CAMRAD II UML Diagram. 36
5.6 TAU Library Folder Structure. 37
5.7 Sample TAU Simulation Folder Structure. 39
5.8 CAMRAD II Library Folder Structure. 39
5.9 Sample CAMRAD II Simulation Folder Structure. 39
5.10 New Workflow Sequential UML Diagram. 41
5.11 Simplified New Workflow Sequential UML Diagram. 42

6.1 Toy Example: Simulation Setup. 56
6.2 Toy Example: Pointwise Grid. 56
6.3 Perpendicular Simulation Folder Structure. 58
6.4 Rotor Blade Sections in CAMRAD II 60
6.5 Rotor Blade Aerodynamic Panels in CAMRAD II 60
6.6 Case 1: Rotor Blade Model 61
6.7 Modified Delta Airloads Algorithm Workflow 62
6.8 CAMRAD II Output File Excerpt. 67
6.9 Json Handshaking Procedure 68
6.10 Failed Mapping of Aerodynamic Loading Data 70
6.11 Successful Mapping of Aerodynamic Loading Data 71

6.12 Comparison of Nearest-Neighbor Interpolation and RBF Interpolation. 72
6.13 Comparison of Full and Piece-wise Data Passing. 73
6.14 Passing of Full and Piece-wise Force Data in x- (left), y- (middle) and z-axis (right). 73
6.15 Relative Error of Full and Piece-wise Force Data in x- (left), y- (middle) and z-axis (right) 73
6.16 Visualization of preCICE piece-wise configuration file. 75
6.17 Motion Hierarchy of the 4-bladed CFD rotor blade simulation. . 78
6.18 Isolated Rotor Blade Simulation with Deformation Folder Structure. 80

7.1 Snapshot of TAU-CalculiX Toy Example. 81
7.2 Displacements of watchpoint tracked in TAU-CalculiX and OpenFOAM-CalculiX. 82
7.3 Flowfield Around Rotor Blade. 82
7.4 Comparison of TAU and CAMRAD II thrust distribution (F_z) prior to coupling. 83
7.5 Comparison of TAU and CAMRAD II aerodynamic force and moment data prior to coupling. 83
7.6 Comparison of TAU and CAMRAD II thrust distribution (F_z) after 4 coupling iterations. 84
7.7 Comparison of TAU and CAMRAD II aerodynamic force and moment data after 4 coupling iterations. 84
7.8 Case 1: Profile Analysis 87
7.9 Comparison of CAMRAD II collocation point position and calculated collocation point position. 89
7.10 Comparison of calculated collocation point position and TAU deformation + motion. 90
7.11 Average Relative Error of Deformation Case 91

A.1 Basic TAU Simulation Folder Structure. 106

List of Tables

6.1 Rotor geometry and test conditions. 60

7.1 Overview of CAMRAD II control values per coupling iteration. . 85
7.2 Profile of original script. 86
7.3 Profile of TAU adapter script. 86
7.4 Profile of CAMRAD II adapter script. 86

B.1 Parameters for TAU Config class 115
B.2 Parameters for TAU Interface class 115
B.3 Parameters for TAU Simulation class 115
B.4 Parameters for CAMRAD II Config class 117
B.5 Parameters for CAMRAD II Interface class 118
B.6 Parameters for CAMRAD II Simulation class 119
B.7 Parameters for CAMRAD II RotorInfo class 119

C.1 Member Attributes of TAU DataHandler class 125

List of Algorithms

1 Serial-explicit coupling scheme algorithm 25
2 Parallel-explicit coupling scheme algorithm 25
3 Serial-implicit coupling scheme algorithm 26
4 Parallel-implicit coupling scheme algorithm 26
5 Original coupling algorithm 32
6 TAU-Python algorithm for tightly-coupled case 57
7 CAMRAD II algorithm for loosely-coupled case 63
8 TAU-Python algorithm for loosely-coupled case 65
9 TAU-Python algorithm for loosely-coupled case with deformation 77

Nomenclature

Notation

symbol	description	unit
c	blade chord length	m
M_x	rotor hub rolling moment (pos. forward)	Nm
M_y	rotor hub pitching moment (pos. right)	Nm
N_b	number of blades	-
R	rotor radius	m
r_tw	zero twist radial station	-
t	time	s
T	rotor thrust	N
T_∞	air temperature	°C
V_∞	free stream velocity	m/s
θ	pitch angle	deg
θ_S	rotor shaft pitch angle	deg
Θ_tw	linear blade twist per span	deg
μ	advance ratio	-
ρ	air density	kg/m^3
σ	rotor solidity, $\sigma = N_\mathrm{b}c/(\pi R^2)$	-
ϕ	flap angle	deg
ϕ_S	rotor shaft roll angle	deg
ψ	azimuth angle, lag angle	deg
ψ_t	rotor unsteady rotation angle, $\psi_t = \omega t$	deg
ω	rotational speed	rad/s

Acronyms

AFDD	US Army Aeroflightdynamics Directorate
API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
BVI	blade-vortex interaction
CAMRAD II	Comprehensive Analytical Model of Rotorcraft Aerodynamics and Dynamics

CFD	computational fluid dynamics
CHANCE	Complete Helicopter Advanced Computational Environment
CSD	computational structural dynamics
DLR	German Aerospace Center *(Deutsches Zentrum für Luft- und Raumfahrt)*
DNW	Deutsch-Niederländische Windkanäle
FPR	Full Potential Rotor
FSI	Fluid-Structure Interaction
HART	Higher-Harmonic Control Aeroacoustics Rotor Test
HOST	Helicopter Overall Simulation Tool
HPC	High Performance Computing
NASA	National Aeronautics and Space Administration
ONERA	Office National d'Etudes et de Recherches Aérospatiales
OOP	Object-Oriented Programming
preCICE	Precise Code Interaction Coupling Environment
RBF	Radial Basis Function
TUM	Technical University of Munich
UML	Unified Modeling Language
WAVES	Without Artificial Viscosity Euler Solver

1. Introduction

1.1 Motivation

The accurate simulation of a rotor blade flowfield is a complex task—requiring the consideration of the aerodynamics, motion and deformation of the rotor blade. As these phenomena are strongly interlinked, this is not easy. One of the major difficulties in simulating such a complex multi-physics problem is the development of a high-fidelity solver that is able to consider both complex phenomena in their entireties, which is no mean feat.

The current paradigm is to employ two separate solvers or simulation *participants*, each adept at solving for a single physical phenomenon in the multi-physics problem. In this work, the coupling of two specialised solvers is of interest: a computational structural dynamics (CSD) solver and a computational fluid dynamics (CFD) solver. The CSD solver is Comprehensive Analytical Model of Rotorcraft Aerodynamics and Dynamics (CAMRAD II), developed by Johnson Aeronautics. The CFD solver is TAU, developed by German Aerospace Center *(Deutsches Zentrum für Luft- und Raumfahrt)* (DLR). TAU is highly specialized in solving fluid problems around complex geometries, while CAMRAD II describes elastic blade behavior and provides trimmed solutions.

While a rudimentary coupling of these two solvers has been developed at the Chair of Helicopter Technologies [1], the coupling is highly specialised for a single simulation case. If this methodology would be used for future couplings, a unique coupling adapter would be required for each simulation case. This would be undesirable as each coupled simulations would need to be maintained independently and new coupled simulations cannot easily exploit old code. This problem is further exacerbated by the high customizability of TAU solvers, which are typically written in the `TAU-Python` Application Programming Interface (API) provided by DLR.

Hence, this work focuses on the creation of a *general* coupling framework that supports the "plug-and-play" of new solvers. The development and imple-

mentation of such a generalized framework can be split into two teps:

1. The development of a modular code structure that retains the easy customizability provided by the `TAU-Python` API

2. The developement of a data exchange interface that minimizes code interdependencies between two separate solvers

The first step is handled via an Object-Oriented Programming (OOP) approach by delegating portions of the general `TAU-Python` simulation code into distinct classes that allow sub-classing. This approach isolates the interfaces between separate solvers and allows the easy customization of critical segments of the simulation work-flow.

The second step incorporates the Precise Code Interaction Coupling Environment (preCICE) coupling library to oversee and control data passed between simulation participants during a multi-physics simulation. preCICE employs a "black-box" approach in coupling solvers and provides sophisticated support for passing information along shared boundaries.

This thesis provides a preCICE adapter that supports custom TAU solvers using the `TAU-Python` API and a Python-based preCICE adapter for CAMRAD II. Three tutorial cases are also implemented in this thesis: a simple example tutorial example from the preCICE respository, the existing coupled simulation developed at the Chair of Helicopter Technologies and an extension of the existing coupling to include grid deformation.

1.2 Structure of Work

This work is organized into eight distinct chapters. After the introductory chapter, the second chapter gives a literature review of the previous work in the field. Key concepts relevant to the work are introduced in Chapter 3, including coupling strategies, an overview of the fluid and solid solver numerical solvers and an introduction to Radial Basis Function (RBF) interpolation, which is used when deforming grids. Chapter 4 introduces the software tools used in this thesis, namely TAU, CAMRAD II and preCICE. Chapter 5 describes the areas of possible improvement in the previous coupling, outlines the new code structure addressing these issues and illustrates the workflows of the newly introduced TAU and CAMRAD II adapters. Chapter 6 describes the implementation of the three tutorial cases in greater detail, providing background information of the various challenges faced and the resultant solutions. Chapter 7 examines

the results of the coupled simulations. Finally, chapter 8 provides an overview of the results, summarizes the most important points of this work and suggests possible future improvements to the developed adapter code.

2. Literature Review

This chapter provides a short overview of the various advances in the coupling of CSD and CFD solvers as related to the field of rotorcraft simulations.

As mentioned in the introduction, the coupling of a solid and fluid solver is not a novel concept. However, the very first multi-physics simulations in the field of rotorcraft simulations typically used a single solver. CSD solvers employing Finite Element techniques were successfully used to model rotor blade structural dynamics and control during rotorcraft operations [3]. These CSD solvers typically incorporated simplified aerodynamics models to calculate the aerodynamic loads for the problem to reduce computational costs [4]. Even today, widely used Solid Dynamics solvers (such as CAMRAD II, RCAS and DYMORE) do not inherently support highly-complex aerodynamic models due to the extreme inherent difficulty of such an undertaking [5, 6, 7].

While tightly-coupled solvers were widely regarded as the more accurate approach (see Section 3.2 for further details), hardware limitations forced the development of computationally cheaper coupling methodologies. One of the earliest coupling implementations include the loosely-coupled transonic simulations developed by Tung et al. in 1986 that coupled CAMRAD with the Full Potential Rotor (FPR) solver [8]. This work was followed by a series of loosely-coupled solvers using similar techniques [9, 10, 11].

With advancements in hardware computational capabilities in the early 1990s came the possibility of using Navier-Stokes or Euler solvers in coupled simulations. Bauchau et al. developed a proof-of-concept tightly-coupled solver coupling CAMRAD/JA and OVERFLOW to simulate isolated UH-60A Blackhawk rotor blades [12, 13, 14]. They also developed a tightly-coupled solver using CAMRAD/JA and FPR to investigate the improvements of using a Euler solver to replace the typical lifting-line aerodynamics model using the Puma helicopter test results [15].

The next years saw the introduction of the *delta airloads method* used by Pahlke et al., Servera et al. and Potsdam et al. [16, 17, 18]. Pahlke et al. and

Servera et al. both investigated the 7A and 7AD rotors [16, 17]. Pahlke et al. used a loose coupling of FLOWer and the DLR CSD code S4 and Servera et al. coupled Helicopter Overall Simulation Tool (HOST) and Without Artificial Viscosity Euler Solver (WAVES) [16, 17]. Potsdam et al. used a loose coupling of CAMRAD II and OVERFLOW-D [18].

Altmikus et al. provided a comprehensive direct comparison of loosely- and tightly-coupled rotor blade simulations [19]. The simulation used the framework developed by the Complete Helicopter Advanced Computational Environment (CHANCE) project, coupling the DLR FLOWer CFD code and the EUROCOPTER flight mechanics tool HOST to simulate the 7A model rotor developed by ONERA and EUROCOPTER. This study was validated in the ONERA S1MA wind-tunnel. The study determined that the results of the loosely-coupled simulation were comparable with those of the tightly-coupled simulation, accompanying a 2.5 overall decrease in time to solution. [19]

More investigations using the delta airloads method continued through the mid 2000's with Abras et al. experimenting with the use of unstructured grids in loosely-coupled simulations of the UH-60A rotors [20]. They demonstrated that these methods produced results comparable to uncoupled structured overset code [20]. A series of NASA funded wind tunnel tests were run in May 2010 and the results were used as validation data for a series of loosely-coupled simulations: Marpu et al. coupled a hybrid CFD solver, GT-Hybrid, with DYMORE [21] and. Lee-Rausch et al. performed their investigations with a FUN3D-CAMRAD II coupling [22].

Loosely-coupled rotor blade simulations have also been used for rotorcraft acoustic analysis or blade-vortex interaction (BVI). Boyd et al. and Lim et al. investigated noise prediction of the Higher-Harmonic Control Aeroacoustics Rotor Test (HART) II and UH-60A rotor using a loosely-coupled OVERFLOWII and CAMRAD II simulation [23, 24].

3. Theoretical Principles

This chapter introduces the various basic theoretical principles required for the remainder of the work, namely the coupling approches for partitioned solvers, coupling approaches in rotor blade simulation, the Chimera meshing technique as well as the mathematical principles behind the RBF interpolation method.

3.1 Monolithic vs. Partitioned Coupling Strategies

One of the earliest methods to couple aerodynamics and structural dynamic equations in rotorcraft simulation was the *monolithic* approach, such as the one proposed by Hübner et al. [25]. With such an approach, a single solver software that incorporates the mathematical models of both rotor blade strucutural dynamics and flow field fluid dynamics would be created. Certainly, such a solver, if well-implemented, would be highly performant, containing inherent internal coupling, a higher degree of robustness, and easier error control [25]. However, the investment required for the development process is significant and most monolithic solvers must be developed fully in-house, as is the case for the work by Geruswamy et al. [13, 14]. Such solvers are also highly specialized and are typically limited to specific domains or sets of problems. Similarly, from an implementation point of view, the upkeep, development and maintenance of such complex, multi-disciplinary code would difficult, as maintainers would need significant expertise across a broad range of domains.

On the other extreme is the *partitioned* approach, where multiple single-physics solvers are *coupled* to solve multi-physics problems. This approach adheres to the computing paradigm of "low coupling, high cohesion"—solvers are run independently and data on shared interfaces is exchanged as needed. The partitioned approach does introduce a degree of complexity when passing information between two independent solvers, which, if handled without care, can introduce mapping errors to the coupled simulation. However, the main benefits of such an approach lies in its modularity—development of solvers can occur in

parallel and maintenance is greatly simplified. As a result, two highly-specialised, state-of-the-art solvers can be combined to exploit each others strengths.

3.2 Tightly-Coupled vs. Loosely-Coupled Rotor Blade Simulations

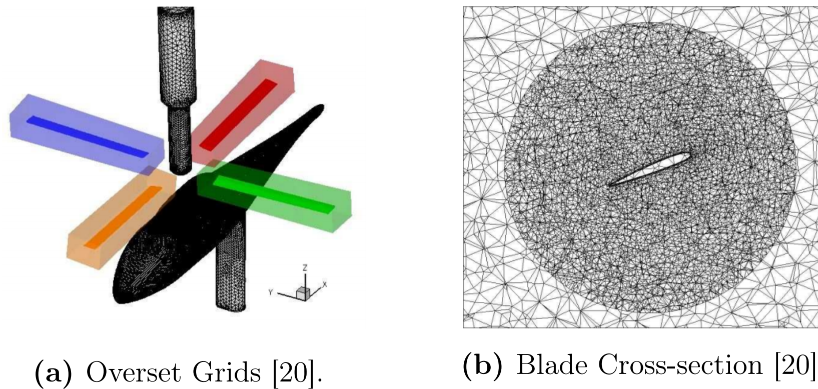
In a coupled rotor blade simulation, there are a further two approaches to exchanging data between simulation participants. The first and more traditional approach is to use a *tightly-coupled* simulation, in which data is exchanged between solvers after each simulation time step [26].

A tightly-coupled approach produces more accurate results as data at the shared boundaries remains time-accurate throughout the simulation (and this approach is still necessary when solving for transient solutions). However, such an approach is deeply involved and is very computationally expensive due to the large volume of data that needs to be processed in each simulation time step. Hence, for problems with a periodic solution, a *loosely-coupled* approach can be employed.

A loosely-coupled simulation only exchanges information periodically, typically after the coupled participants complete one or several full revolutions [17]. Other than the decreased computational cost, such simulations also inherently produce *trimmed* solutions: Given a set of aerodynamic loads, most contemporary rotorcraft CSD software will iteratively determine the input controls which satisfy a trim condition. The CSD solution under these conditions is known as the trimmed solution and is useful to researchers as it reflects the operational conditions of a rotorcraft in a steady-state situation (such as free flight, hover, steady ascent, or descent). Additional steps are required for tightly-coupled simulations to produce trimmed results, such as iteratively restarting the full simulation after modifying the helicopter input controls. All these further add to the computational resource requirements of a tightly-coupled simulation.

3.3 Introduction to Chimera Grids

The fluid solver in this work employs Chimera grids for meshing purposes. Chimera or Overset grid approach is an adaptive meshing strategy first introduced by Benek et al. in 1986 [27]. This technique was designed for use in the aerospace field and resolves several difficulties when generating meshes for rotorcraft simulations.



(a) Overset Grids [20]. (b) Blade Cross-section [20].

Figure 3.1: Chimera Grid Example showing Chimera Block and Cross-section.

The Chimera grid strategy performs domain decomposition, allowing for easier fine-mesh generation around areas of interest and a coarser mesh in the farfield regions. One example is shown in Fig. 3.1, where four differently coloured subdomains are *embedded* into the farfield mesh. These embedded meshes are allowed to follow the movement of these areas of interest through the domain, i.e. the embedded meshes can follow the movement of rotor blades through the farfield mesh.

Similar domain decomposition strategies prior to this technique provide similar benefits but require gridlines to be continuous across subdomain boundaries [28, 29]. Chimera grids take a different approach: subdomains overlap at shared boundaries and data along shared boundaries is interpolated from these shared regions [27]. This can be seen in Fig. 3.1b, where the finer embedded mesh overlaps with the coarser farfield mesh.

Additionally, Chimera grid approach supports adaptive meshing, in which nodes in subdomains are modified depending on the state of the equations, effectively clustering points around regions of high gradients. Hence, initial Chimera grids do not need to fit the simulated model completely but instead are adapted throughout the course of the fluid simulation. [27]

3.4 Introduction to Radial Basis Functions

The deformation executable used in this work to perform grid defomation employs RBF interpolation. This interpolation method maps data between two distinct grids or discretisations [30]. The general principle behind RBF interpolation is simple; given an arbitrary discretisation $x_i \in \mathbb{R}^d$ of a given domain with arbitrary nodal values, we assume that each point has a decaying “influence” based solely on a function ϕ of radial distance $|x - x_i|$—known as a *radial basis function*.

Then, the value of any arbitrary point in the given domain can be interpolated by a linear combination of all "influences" in the domain [31]. Given a function $f : \mathbb{R}^d \rightarrow \mathbb{R}$ that determines nodal values based on nodal position, an RBF interpolant $S : \mathbb{R}^d \rightarrow \mathbb{R}$ approximating f can be represented by Eq. 3.1.

$$S(x) = \sum_{i=1}^{n} \gamma_i \phi(x - x_i) \tag{3.1}$$

In this work, one possible function ϕ, the Gaussian RBF, is the default function used in the TAU grid deformation executable, as seen in Eq. 3.2 [32]. Other RBFs include the original Multi Quadratic RBF, Gaussian and Thin Plate Splines [30, 31].

$$\phi(x - x_i) = e^{(-(s||x-x_i||)^2)} \tag{3.2}$$

However, considering every single grid point when performing this interpolation is an $O(N^3)$ algorithm, and this is highly expensive. Thus, considering that the "influence" of distant points is close to zero, an RBF *cut-off radius* can be implemented [30]. Points beyond this cut-off radius are not considered when performing the linear combination. The modified linear combination ϕ_{w} is seen in Eq. 3.3, where the linear combination for point $y \in \mathbb{R}^d$ is calculated with a cut-off radius of r [30].

$$\phi_{\mathrm{w}}(x_i) = \begin{cases} 0 & , ||x_i - y|| > r \\ \phi(x_i) & , ||x_i - y|| \leq r \end{cases} \tag{3.3}$$

4. Tools

There are three main softwares used in the coupling: TAU, CAMRAD II and the preCICE library. This section gives a brief overview of each tool used, how they function and the ways by which the adapter code interacts with each software.

4.1 TAU

TAU is a CFD tool developed by DLR in Göttingen in the 1990s to solve fluid-flow problems around complex geometries, ranging from the subsonic to hypersonic regimes [33, 34, 35]. The software, while mainly used for internal projects at DLR, has also been used in several national and international projects, such as MEGAFLOW, FLOMANIA and DESider [36, 37, 38, 39] and boasts high parallel efficiency for High Performance Computing (HPC) purposes [40, 41, 42].

The TAU software is not a single all-encompassing code, instead it employs a modular approach, providing separate stand-alone executables for each functionality. For this work, the modules for grid-partitioning or `preprocessing`, turbulent 1-equation solver or `turb1eq` and grid-deformation or `deformation` are used.

One of the main limitations of developing wrapper code for TAU is that the source code is not open-source. Thus, direct interaction with the source code is not possible. Instead, a Python interface, `TAU-Python`, is used to interact with the TAU software. The adapter code uses this interface; further details are provided in Chapter 5 and 6.

4.1.1 TAU Simulations

As mentioned in the previous section, the TAU software consists of multiple stand-alone executables. Each of these executables is controlled by a single parameter file or *parafile*. The parafile contains key-value pairs of significant input parameters; a sample of these key-value pairs can be seen in Fig. 4.1.

These key-value pairs can be divided into two broad categories: variables

Parafile Excerpt

```
Files/IO ------------------------------------------
       Boundary mapping filename: ./grid/boundary.bmap
           Primary grid filename: ./grid/grid_primary
         New primary grid prefix: ./grid_deform/deform.grid
                     Grid prefix: ./grid/dg/grid.dg

Output definition --------------------------------
   Field output description file: (thisfile)
             Output files prefix: ./output/heli
             Field output values: Rrho_cp_mach_blank_xyzbody

 Surface output description file: (thisfile)
           Surface output values: xyz_fxyz_p_cp
           Surface output period: 15
```

Figure 4.1: TAU Parafile Excerpt.

that control the simulation loop and the paths to input files for the simulation. The former includes variables such as the number of inner iterations per solution time step, the Reynold's number or the period between surface/field solution outputs. The latter points to input files that define the simulation (such as grid files, partitioned or primary, that define the fluid mesh in serial or parallel execution); motion files defining the movement of blades during the simulation; text files defining point by point deformation (also referred to as *scatfiles*) and output file locations. These are explained in greate detail in the TAU user guide documentation [32]. Prior to discussing the handling of these inputs and outputs, the following section briefly introduces the capabilities of the `TAU-Python` API.

4.1.2 TAU-Python API

The `TAU-Python` API allows the user to control the TAU simulation via specific classes, each class analogous to a TAU stand-alone executable. Prior to initializing these classes, the user must first initialize the `TAU-Python` environment to ensure that underlying `TAU` processes are initialized and the global `TAU-Python` methods work as intended (such as the TAU steering methods `tau_mpi_rank()`, `tau_mpi_nranks()` and `tau_parallel_sync()` for control of parallel executions).

Then, the various `TAU-Python` classes can be initialized. For this work, only the following four TAU classes are used, namely:

1. PyPara: This class extracts data from a given parafile and allows run-time

modifications to parafile contents, effectively modfying simulation input variables at runtime.

2. PyPrep: This class controls the ptau3d.preprocessing stand-alone executable. The executable controls the partitioning of the main grid in the case of a parallel use-case.

3. PySolv: This class controls the ptau3d.turb1eq executable that runs the simulation loop.

4. PyDeform: This class controls the deformation executable that creates a deformed grid based off the primary gridfile and a user provided scatfile

After the completion of the TAU simulation loop, certain cleanup methods should be called to ensure no memory leaks, namely the `PySolv.finalize()` method and the `exit('TAU')` global method.

A sanitised basic execution loop is shown in Listing 4.1.

Listing 4.1: Simple TAU Simulation Loop.

```
import tau_python

""" Setup tau_init_variables and parafile_path """

# Initialize TAU environment
tau_python.init(tau_init_arguments)

para = PyPara(parafile_path)
prep = PyPrep(parafile_path)
solv = PySolv(parafile_path)
deform = PyDeform(parafile_path)

# Preprocess grid for parallel execution
prep.run(write_dualgrid=1, free_primgrid=1, verbose=1)

# Get number of unsteady time steps from parafile
unsteady_steps = para.get_para_value("Unsteady Timesteps")

# Initialize TAU solver
solv.init()

for i in range(unsteady_steps):
    # Run single outer loop step
    solv.outer_loop()
    # Write solution files per parafile output period
    solv.output()
    # Deform grid based on scatfile in parafile
    deform.run(read_primgrid=1, write_primgrid=1)
    tau_python.tau_parallel_sync()

# Finalize solver environment
solv.finalize()
# Cleanup of TAU-Python environment
tau_python.tau('exit')
```

Note that the simulation algorithms used in this work are relatively straightforward and the extensive `TAU-Python` API allows for simulation loops of far greater complexity. Additional classes allow, for example, for the reading of raw TAU data from memory buffers or allow the creation of plt graphs (used to visualize solutions) during runtime. This flexibility is a factor that must be taken into consideration when designing the framework for the TAU adapter.

After the creation of a simulation loop, the next step is to modify simulation inputs and outputs to facilitate coupling between the TAU solver and the coupled solid solver.

4.1.3 TAU Data Input

This subsection deals with how TAU receives data input from solid solvers in a coupled simulation. The two main inputs addressed in this work are the blade motion and the deformation data received from CAMRAD II.

Motion Input

The motion of each component in a TAU simulation is defined by the motion hierarchy file and motion definition file prescribed in the parafile. The motion hierarchy in TAU defines specific nodes (where motion occurs) and the dependencies between each node, shown in Fig. 4.2.

```
Motion Hierarchy

                    Node name: a
          Node reference frame: inertial
      Node controls grid block: -1
  Node motion description id: a_id
                       hdf end

                    Node name: b
          Node reference frame: a
      Node controls grid block: 1
  Node motion description id: b_id
                       hdf end
```

Figure 4.2: TAU Motion Hierarchy Excerpt.

The figure shows the definition of two nodes “a” and “b”. “a” is the root node and takes reference from the inertial reference frame with additional motion defined as “a_id” in the motion definition data. “b” takes reference from “a” (i.e. moves with “a”) with additional motion defined as “b_id”. We see from the

Motion Definition

```
                    Motion description id: a_id
                         Type of movement: periodic
         Origin of local coordinate system: 0 0 0
         Degree of polynomial for rotation: 1
  Polynomial coefficients for rotation yaw: 0 -6245.24524

            Reduced frequency for rotation: 3.3333
        Reduced frequency reference length: 1 1 1
           Number of time steps per period: 360
                                   mdf end

                    Motion description id: b_id
                   Hinge - specify vector: 0 -1 0
     Degree of Fourier series for rotation: 1
    Hinge - reduced frequency for rotation: 3.3333
Hinge - reduced frequency reference length: 1
Hinge - Fourier coefficients for rotation (cos): 2.0525 -0.0014802
Hinge - Fourier coefficients for rotation (sin): 0 0.0022614
                                        mdf end
```

Figure 4.3: TAU Motion Hierarchy Excerpt.

motion hierarchy that node "b" controls grid block 1, which references a set of nodes defined in the primary grid. On the other hand, "a" controls no nodes in the grid; this node acts as a simple reference frame.

The motion definition supplements the motion hierarchy and sets the exact motion at each node; a sample excerpt is seen in Fig. 4.3. Motion is defined in blocks and the end of each block is indicated by the string `mdf end` . Within a block, motion can be defined via a polynomial. For example, in the motion definition block `a_id`, "a" periodically rotates in the yaw direction around coordinates $(0, 0, 0)$ at $-6245.245\,24°/s$. Alternatively, we can provide motion data as a Fourier series: node "b" rotates along hinge vector $(0, -1, 0)$ with an angular deflection calculated via the Fourier series defined in the block.

Deformation Files

TAU deformation is defined in American Standard Code for Information Interchange (ASCII) style text files, referred to as *scatfiles*, which contain deformation locations (where deformation occurs in the grid) and magnitudes (in x-,y- and z-axes) in the TAU-Code-Grid frame. A sample of a scatfile is seen in Fig. 4.4.

The first number, 1200, indicates the total number of deformation locations. Each row then lists the x, y, and z coordinates and deformations in the x-, y- and

Scatfile Excerpt

```
1200
-0.05000 0.48000 -0.00750 -0.00439 -0.00106 -0.01730
-0.04474 0.48000 -0.00750 -0.00440 -0.00102 -0.01741
-0.03947 0.48000 -0.00750 -0.00440 -0.00098 -0.01751
-0.03421 0.48000 -0.00750 -0.00440 -0.00093 -0.01762
-0.02895 0.48000 -0.00750 -0.00440 -0.00089 -0.01772
-0.02368 0.48000 -0.00750 -0.00440 -0.00085 -0.01782
-0.01842 0.48000 -0.00750 -0.00440 -0.00080 -0.01793
```

Figure 4.4: TAU Scatfile Excerpt.

z-axes. For example, at coordinates $(-0.05000.0.48000.-0.00750)$, a deformation of $(-0.00439 - 0.00106 - 0.01730)$ will occur. This deformation is applied to the primary grid listed in the TAU parafile via RBF interpolation, as mentioned in Section 3.4.

4.1.4 TAU Data Output

This subsection deals with TAU data output. During the investigation of tight and loose coupling techniques, there were a multitude of methods to access solution data during runtime [43]. The more traditional approach is to access data from output files. Alternatively, there are several more efficient methods that do not involve access data written to disk. Many C/C++ implementations would implement forms of pointer passing or exploit data stored in memory buffers to exchange data between the coupled participants [43].

In TAU, the usage of both methods is possible. TAU writes solution data to disk in the `netCDF` format, which is supported in Pythonvia the `netCDF4` library. The data is stored in a Python `dict` object. Additionally, the `TAU-Python` API also supports the use of memory buffers, which allow access to solution data during runtime.

Certainly, the latter method is ideal. However, the methods employed in this work are strongly constrained by the capabilities of the `TAU-Python` API. The methods used to buffer solution data in memory only have access to data in the current time step. This poses a problem in a loosely-coupled simulation, where data from multiple time steps is passed at each coupling step. If a coupled simulation is restarted just before the coupling step occurs, the `TAU-Python` API would not be able to provide data from previous time steps. In order to deal with this situation, the data exchange must still rely on data written to

disk.

4.2 CAMRAD II

CAMRAD II was developed by Johnson Aeronautics in 1992 to provide an aeronautical analysis tool; the software supports multi-body dynamics with a finite-elements scheme using non-linear elements in a trim model [4, 5]. This software, along with similar CSD codes, such as RCAS and DYMORE [6, 7], is one of the most widespread software solutions for structural dynamics codes in the field of rotorcraft technologies.

4.2.1 CAMRAD II Simulations

The CAMRAD II software functions by splitting a complex aeromechanical system into multiple *pieces*, categorized in the CAMRAD II documentation as environmental, physical and logical pieces [44].

The environmental pieces define the simulation case, such as forward flight, hover, ascent or descent; system operation (which includes the system environment, such as the earth axes, density, operating altitude etc.); wind conditions (including wind speeds, direction and gust axes); operating conditions of the physical system (which defines flight speed, orientation of the various frames of reference, turning rate and physical system constraints); and the periods, which define the period of rotation for the various components of the physical system. [44]

The physical pieces define the physical system components, such as the rotor blades, fuselage, drivetrain etc. CAMRAD II defines a list of supported structure *classes*. These are listed in the CAMRAD II documentation [44] and define the required geometrical definitions as well as the various relevant inter-component couplings. The physical input also defines the required solution data, such as position, force and pressure values and specific sensor locations along the simulated component where such output is of interest. [44]

The logical pieces define the procedure for solving simulation equations; more specifically, they describe the algorithm used to reach convergence. CAMRAD II supports various algorithms, such as implicit time-stepping methods or Newton-Raphson methods, which are applicable for both time or harmonic finite element problems. [44]

The CAMRAD II executable accepts input via shell or a series of tables defining the various simulation pieces listed above. The shell method allows quick

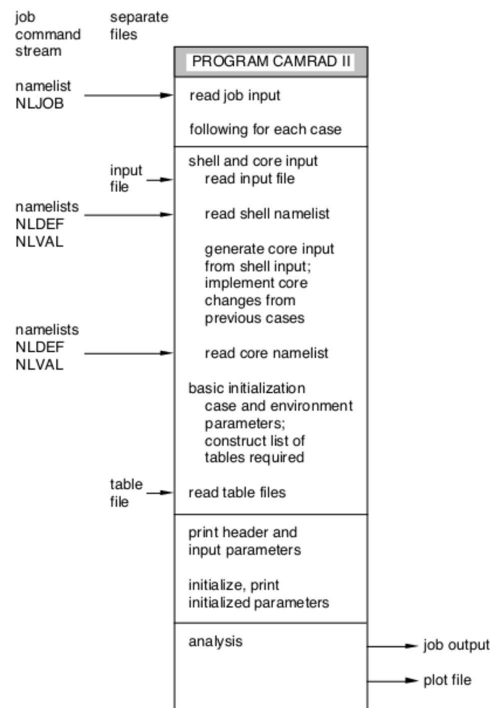


Figure 4.5: CAMRAD II Jobfile Structure [5].

input of various variables but lacks flexibility. Hence, the typical method of choice is via tables, in which an ASCII *jobfile* lists the various input parameter identifiers with their values. A typical jobfile structure can be seen in Fig. 4.5. CAMRAD II accepts jobfiles in the form of Standard, C81 or CAMRAD style tables. The definition of each type of table format can be found in the CAMRAD II user guide [44].

The CAMRAD II software provides a lifting-line aerodynamics model, which assumes a high-aspect ratio of the wing. Thus, the rotor blade aerodynamics problem can be split into an inner and an outer problem, described as the wing and wake problems [44]. The outer problem or the wake problem is solved to determine the angle of attack for a predefined span or *panel* along the wing. The angle of attack then utilizes a look-up table correlating angle of attack to lift, drag and moment coefficients specific to the simulated wing to determine aerodynamic loads. In this work, we supplement the aerodynamic loads from this model with loads generated by the TAU fluid solver.

Noting that the software is proprietary, access to the CAMRAD II source code is not possible. As a result, we wrap the CAMRAD II software with a Python script that parses CAMRAD II ouput data and post-processes aerodynamic load data from the coupled fluid solver. Updated aerodynamic loading data

is provided as a series of input loading tables acceptable by the CAMRAD II executable during start-up. A more detail description of this process is provided in the next subsection, Section 4.2.2.

4.2.2 CAMRAD II Data Input

This subsection describes how CAMRAD II receives data from fluid solvers in a coupled simulation. This work mainly deals with incorporating aerodynamic load data into CAMRAD II's trimmed simulations.

4.2.3 Force Tables

CAMRAD II accepts external aerodynamic loads in the form of ASCII *delta tables*, which add a correction to the output of its internal lifting line aerodynamic solver. A sample force table is shown in Fig. 4.6. The type of data is indicated by the term `FQCX` (meaning the x-component of force along the rotor blade quarter-chord position) in the top left corner.

CAMRAD II Force Table Excerpt

```
21 25 FQCX
              0.00           15.00          30.00          45.00 ...
              120.00         135.00         150.00         165.00 ...
              240.00         255.00         270.00         285.00 ...

0.24          4.502272E-01   3.694587E-01   2.853749E-01 ...
              1.469075E-01   2.010710E-01   2.555786E-01 ...
              8.717851E-02   1.334834E-01   2.308048E-01 ...
              4.502272E-01

0.28          5.609823E-01   5.179287E-01   4.461996E-01 ...
              3.379185E-01   3.835550E-01   4.232441E-01 ...
              1.091121E-01   1.148474E-01   1.850540E-01 ...
              5.609823E-01
```

Figure 4.6: CAMRAD II Force Table Excerpt.

The two numbers in front of the type of data, 21 and 25 indicate the spatial and temporal discretisation of the correction values respectively. These correctional forces and moments are applied at the 21 sensor positions along the rotor blade for each of the 25 azimuth positions in a single revolution. Each data value (at a given time and position) indicates the difference between the calculated CAMRAD II aerodynamic loads and the aerodynamic loads provided by the

external fluid solver. Note that the lines in these ASCII files wrap around such that data from a single collocation point can span multiple lines—this must be considered when updating or writing these delta table files.

4.2.4 CAMRAD II Data Output

CAMRAD II output data is written to disk in large ASCII text files. Unlike in TAU, no interface exists for the CAMRAD II standalone executable. Hence, data must be read from output files. A custom regex parser for the CAMRAD II output files was introduced to improve the original CAMRAD II parser [1].

Recalling that CAMRAD II produces trimmed solutions (refer to Section 3.2 for details) for whole revolutions, data in CAMRAD II simulations is written in two forms: as time-discretized values spaced 15° apart or as Fourier series. This work reads and passes the Fourier series information during coupling as TAU can accept motion data as Fourier series coefficients. Similarly, this greatly reduces the amount data that needs to be passed in each coupling step.

4.3 TAU & CAMRAD II Coordinate Systems

Before discussing the coupling library, it is pertinent to discuss the coordinate systems used when handling the data. Both TAU and CAMRAD II present solution data in a variety of coordinate systems and the data from these systems must be consistent in order to map the correct data values when coupling.

Fig. 4.7 displays the four coordinate systems used by TAU. The TAU-Code-Grid frame is the coordinate system defined in the primary grid file. The Body-fixed frame is similar to the TAU-Code-Grid, other than the fact that the origin is defined by the user in the parafile and that the x and z axis are flipped. Both axes rotate with the motion of the body and do not change as the simulation progresses. The Geodesic frame does not rotate with the body and is the inertial frame of the system. The Aerodynamic frame is, effectively, a rotated Geodesic frame such that the x axis is parallel with the flight velocity vector V_∞. In this work, the velocity vector is already parallel to the Geodesic frame x axis and, thus, the two frames are identical. [45]

CAMRAD II also offers several reference frames for solution data. However, only two are relevant to this work: the Wing-fixed axes and the Inertial axes. These two axes are analogous to the Body-fixed frame and Geodesic frame in TAU, shown in Fig. 4.7 and can be similarly customized in the input tables. However, the Body-fixed axes in CAMRAD II flips the x and z axis of the

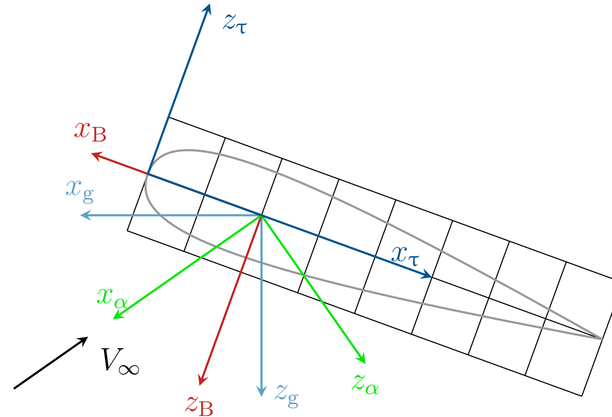
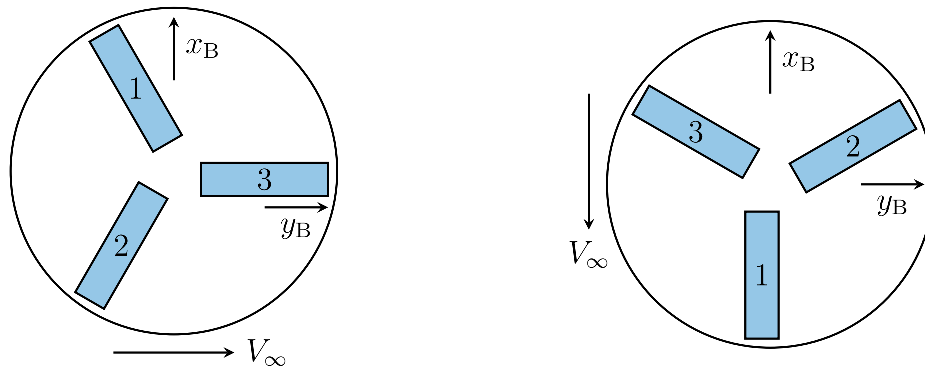


Figure 4.7: Coordinate Definitions used in TAU. τ: TAU-Code-Grid frame, B: Body-fixed frame, g: Geodesic frame, α: Aerodynamic frame. Adapted from the TAU User Gudie [45].

Body-fixed frame in TAU—which must be kept in mind when processing data.

A second consideration when passing data between these two packages is blade alignment. The conventions in TAU and CAMRAD II can differ and care must be taken to ensure that the correct blades are referenced. For example, it is possible that the blades are oriented as in Fig. 4.8 in the respective Body-fixed frame in TAU and the Body-fixed axes in CAMRAD II



(a) TAU Convention: Rotor with reference blade (1) aligned with Body-fixed axes.

(b) CAMRAD II Convention: Rotor with reference blade (1) pointing away from incoming flow velocity (CAMRAD II convention).

Figure 4.8: Possible differences in axes convention shown in (a) and (b) [1].

In this situation, there are two issues when passing data in this coupling:

1. The orientation of the axes in TAU and CAMRAD II differ
2. The referenced blades are not the same: Blade #1 in TAU refers to Blade #2 in CAMRAD II

Thus, these factors must be considered when handling data from each solver. A standard orientation must be defined for each coupling.

4.4 preCICE

preCICE is a coupling library developed at the Technical University of Munich (TUM), the University of Stuttgart and the University of Erlangen. The library is designed to support multi-physics simulations by coupling multiple single-physics solvers using a partitioned, "black-box" approach. Using this approach, preCICE requires no information about the inner workings of either solver. Instead, preCICE exchanges information at user-defined interfaces shared by the coupled participants while minimizing interference with their source code or internal operations. The pre- and post-processing of exchanged data is performed by an *adapter*, which ensures that data is processed and formatted to a given standard defined by the specific coupling, sets the time step and implements checkpointing as necessary.

For the purposes of this work, this coupling library is chosen for several reasons. Firstly, one of the main deliverables of the work is the modularity of the final product. The "black-box" approach of preCICE allows the multiple participants to work completely independently—any coupled component can then be switched out or replaced with newer, more performant or more efficient solvers.

The second advantage of preCICE is passing data between non-identical or partitioned grids. preCICE supports multiple methods for data mapping, described in Section 4.4.1.

Additionally, preCICE can be used to control the simulation loops of the various participants and ensures that equation coupling occurs at the correct time steps. This is particularly useful when coupled solvers have vastly different time steps or if the implicit coupling schemes described in Section 3.2 are used.

The last advantage of using preCICE is ease of use. preCICE is designed to be inserted into existing code and the various components of the library can be placed in a typical simulation loop. An example is provided in Section 4.4.3. However, before moving to the details of the preCICE API, we should first examine how data is mapped between shared interfaces.

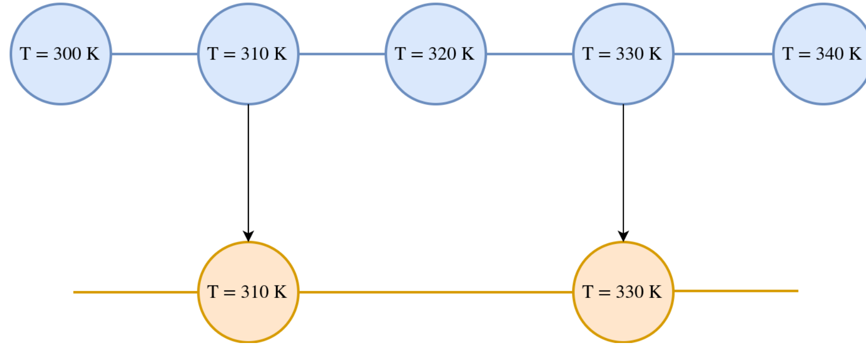


Figure 4.9: Consistent Mapping of Temperatures [46].

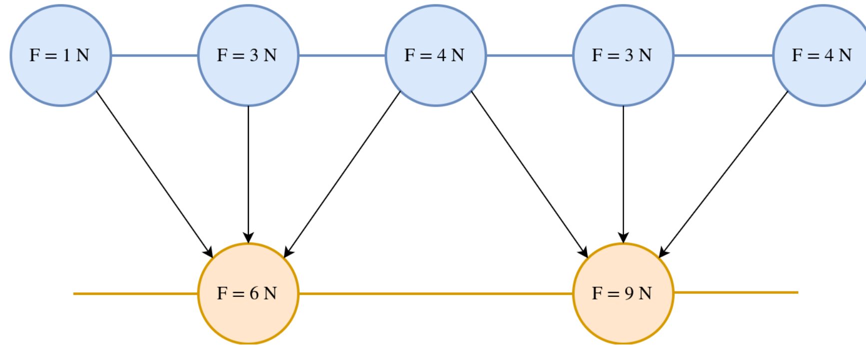


Figure 4.10: Conservative Mapping of Forces [46].

4.4.1 Data Mapping

The preCICE library supports both *conservative* and *consistent* data mappings along shared interfaces with differing discretizations via a variety of methodologies. Consistent mapping ensures that point values between the two discretisations are consistent, as shown in Fig. 4.9. This form of mapping is used for data that is non-conservative, such as temperature, pressure or displacement. Conservative mapping, on the other hand, ensures that the integral values on the shared surface are conserved, as shown in Fig. 4.10. This form of mapping is used for data that must be conserved, such as force or moment data.

These interpolations are linear and can be represented as a matrix product, shown in Eq. 4.1.

$$\tilde{X}_A = M_{AB}X_B, \tilde{X}_A \in \mathbb{R}^n, X_B \in \mathbb{R}^m, M_{AB} \in \mathbb{R}^{nxm} \tag{4.1}$$

In Eq. 4.1, we have a mapping matrix M_{AB} representing a mapping $\Gamma_B \rightarrow \Gamma_A$, M_{AB}, with initial values X_B and interpolated values $\tilde{X}_A$. The mapping is consistent if the sum of all rows in the mapping matrix M_{AB} sum up to one [47], or:

$$\sum (M_{AB})_i j_{i=1}^{n} = 1, \forall j = 1, ..., m. \tag{4.2}$$

A mapping is conservative if the sum of all columns in the mapping matrix

sum up to one [47], or:

$$\sum (M_{AB})_i j_{j=1}^{m} = 1, \forall i = 1, ..., m. \tag{4.3}$$

Hence, a consistent mapping induces a conservative mapping, such that:

$$(M_{AB})_{conservative} = (M_{BA})^T_{consistent}. \tag{4.4}$$

There are currently three different mapping methodologies supported by preCICE. The following descriptions provide a explanation of how each algorithm is implemented in a consistent manner, given a mapping $\Gamma_B \rightarrow \Gamma_A$:

1. **Nearest Neighbor**: Each point in Γ_A finds the nearest point in Γ_B via an arbitrary norm and directly copies the value of this "nearest neighbor" point. This is a simple first order method that does not require additional topological information [47].

2. **Nearest Projection**: Each point in Γ_A determines its orthogonal projection onto Γ_B. The value at this point can be determined via linear interpolation of point values in the relevant cell or face in Γ_B. Note that such an interpolation then requires connectivity data of the discretizations. This method is of second-order if the orthogonal distance between the projected point from Γ_A and the points in Γ_B are significantly smaller than cell size [47].

3. **RBF**: The RBF interpolation is described in Section 3.4.

Note that the conservative mappings for each method are induced by the existence of the consistent mapping. A more in-depth look into the various implenentations can be found in the dissertation of Benjamin Uekerman and the paper by Lindner et al. [47, 30].

4.4.2 preCICE Coupling

Other than how data is mapped between shared interfaces, the when is also similarly important. A coupling scheme, as defined in preCICE, determines this interaction between coupled solvers. There are two variables which control the interaction between solvers: if the coupling is run in *serial* or in *parallel* and if time-stepping of the coupled simulation occurs *explicitly* or *implicitly*. Note that, when referring to coupling schemes, the four listed variables have different definitions compared to their classical meanings in informatics or solver terminology. These variables will be explained in the following algorithms. First, we define two solvers which exchange data across a shared interface. Each solver

requires the output of the other solver to calculate results for the next time step, shown in Eq. 4.5.

$$S_1 : B_2 \rightarrow B_1, S_2 : B_1 \rightarrow B_2 \tag{4.5}$$

An *explicit* coupling exchanges boundary data at a set interval without convergence checks of boundary values B. For simplicity, we can assume that S_1 and S_2 have time steps of the same length and exchange data with every time step in a tightly-coupled simulation. We begin by examining the serial-explicit coupling scheme. In a serial scheme, the two solvers have a set order and the one solver completes the computation of a time step before the next solver continues[1]. Given the two solvers defined above, a serial-explicit coupling that runs for n coupling steps is shown by Algorithm 1.

Algorithm 1: Serial-explicit coupling scheme algorithm

for $t = 1, 2, \ldots n$ **do**
  $B_1^{(t+1)} = S_1\left(B_2^{(t)}\right)$
  $B_2^{(t+1)} = S_2\left(B_1^{(t+1)}\right)$

The first solver S_1 uses old data $B_2^{(t)}$ to calculate the updated values $B_1^{(t+1)}$, which are used for the update step of solver S_2. This form of coupling ensures that solver S_2 always receives updated information.

If the explicit scheme is run in *parallel*, however, the two solvers are instead run simultaneously. In this case, the coupling can be seen as Algorithm 2.

Algorithm 2: Parallel-explicit coupling scheme algorithm

for $t = 1, 2, \ldots n$ **do**
  $B_1^{(t+1)} = S_1\left(B_2^{(t)}\right)$
  $B_2^{(t+1)} = S_2\left(B_1^{(t)}\right)$

Unlike in the serial coupling, both solvers S_1 and S_2 use old data from the previous time step. This may increase the number of coupling iterations required for convergence but may decrease the overall time to solution as the two solvers can run at the same time.

While an explicit coupling is the simplest method to couple two solvers, some coupled simulations are often inherently unstable regardless of time step size restrictions. In such cases, an implicit coupling scheme can be employed, where a fixed-point iteration of the boundary values takes place. While the general

[1] This does not mean that the individual solvers are run with only one process per the classical informatics definition of “serial”.

principles of serial or parallel coupling remain the same, implicit coupling adds a convergence criterion before advancing the time of the coupled simulation see Algorithms 3 and 4.

Algorithm 3: Serial-implicit coupling scheme algorithm

for $t = 1, 2, \ldots n$ **do**
  $converged = False$
  $B_1 = B_1^{(t)}$
  $B_2 = B_2^{(t)}$
  while *not converged* **do**:
    $\tilde{B}_1 = S_1(B_2)$
    $\tilde{B}_2 = S_2(\tilde{B}_1)$
    if $CheckConverged(B_1, \tilde{B}_1, B_2, \tilde{B}_2) == True$ **then**
      $B_1^{(t+1)} = \tilde{B}_1$
      $B_2^{(t+1)} = \tilde{B}_2$
      $converged = True$
    else
      $B_1 = \tilde{B}_1$
      $B_2 = \tilde{B}_2$

Algorithm 4: Parallel-implicit coupling scheme algorithm

for $t = 1, 2, \ldots n$ **do**
  $converged = False$
  $B_1 = B_1^{(t)}$
  $B_2 = B_2^{(t)}$
  while *not converged* **do**:
    $\tilde{B}_1 = S_1(B_2)$
    $\tilde{B}_2 = S_2(B_1)$
    if $CheckConverged(B_1, \tilde{B}_1, B_2, \tilde{B}_2) == True$ **then**
      $B_1^{(t+1)} = \tilde{B}_1$
      $B_2^{(t+1)} = \tilde{B}_2$
      $converged = True$
    else
      $B_1 = \tilde{B}_1$
      $B_2 = \tilde{B}_2$

In these two algorithms, a method $CheckConvergence$ is introduced representing the convergence criterion. If this check fails, the coupling step is repeated using the updated $\tilde{B}_1, \tilde{B}_2$ values. A typical convergence check determines if the change in the coupled data is below a threshold error or $|\tilde{B} - B| < Tolerance$. Note that this is a simplification of preCICE capabilities as the library is not limited to only checking for convergence. preCICE can also insert additional

acceleration methods to speed up convergence of B, such as the *Anderson Acceleration Quasi-Newton*(IQN-ILS) or the *Generalized Broyden Quasi-Newton* (IQN-IMVJ) fixed-point acceleration methods [47]. However, these methods are beyond the scope of this master's thesis.

In summary, preCICE supports four possible coupling schemes, which can be employed when using the adapters developed in this work. They allow the user to run simulations serially or in parallel and allow for fixed-point iteration methods to ensure convergence along shared interfaces. For this work, serial-explicit coupling schemes are employed as prior work has shown that implicit coupling schemes are not necessary for convergence [1].

4.4.3 preCICE API

The preCICE library is written in C++. However, we utilize the Python bindings for preCICE to increase usability. The Python bindings are centered around the `Interface` object and provide a variety of methods to control the coupling between simulation participants. These methods can be classified into three categories: steering methods, auxiliary methods and data access methods.

Steering methods are used to direct preCICE, initializing the preCICE environment, controlling the time stepping of the coupled simulation and terminating the preCICE environment upon completion of the simulation. These methods ensure that coupled solvers with varying time-steps continue to couple at the correct points in time.

Auxiliary methods allow the user to modify coupling logic during execution and act under specific circumstances. These methods are useful in more complex couplings, such as when implicit coupling schemes are used. These methods are not the most relevant to this work as the examined coupled simulations all use explicit coupling schemes.

Data access methods control the passing of data between coupled participants along shared interfaces. preCICE supports the passing of 2D or 3D data along shared interfaces.

A solver can be adapted to couple with other solvers by inserting these methods at specific points in existing solver code. An example is the basic TAU simulation loop (Listing. 4.1), now modified in Listing 4.2. Updated or new lines are highlighted in blue. Note that performing a preCICE coupling for a loosely-coupled simulation is far more involved and will be further explained in Chapter 6.

In this sanitized code snippet, we show the multiple stages required to insert

Listing 4.2: TAU simulation loop with preCICE coupling.

```
import tau_python
import precice
import mpi4py

""" Setup `tau_init_variables`, `parafile_path`, `precice_config`
and simulation `timestep`. Determine variables `mesh_name` and `read_data_name`
and `write_data_name` describing coupled interface name and names of data to be
handled. `rank` and `size` are defined from mpi4mpy.COMM_WORLD"""

# Initialize TAU environment
tau_python.init(tau_init_arguments)

# Setup precice interface participant
precice_interface = precice.Interface(size, rank)
precice_interface.configure(precice_config)

""" Initialize TAU-Python classes"""
# Preprocess grid for parallel execution
prep.run(write_dualgrid=1, free_primgrid=1, verbose=1)
# Initialize TAU solver
solv.init()
""" Define the local discretisation of shared interface `mesh_name`"""
# Setup precice interface
mesh_id = precice_interface.get_mesh_id(mesh_name)
write_data_id = precice_interface.get_data_id(data_name1, mesh_id)
read_data_id = precice_interface.get_data_id(data_name2, mesh_id)
precice_ids = np.zeros(data_size)
precice_interface.set_mesh_vertices(mesh_id, data_size, data_coordinates,
↪  precice_ids)

# Initialize precice interface
precice_interface.initialize()

while precice_interface.coupling_ongoing():
        # Run single outer loop step
        solv.outer_loop()
        # Write solution files per parafile output period
        solv.output()
        """ Store TAU output data in `write_data`, create empty buffer `read_data`
        to store data from other solver """
        # Pass data to solid solver        at end of time step
        precice_interface.write_block_vector_data(write_data_id, data_size,
        ↪  precice_ids, write_data)
        # Advance time for the coupled simulation
        dt = precice_interface.advance(timestep)
        # Pass data to solid solver        at end of time step
        precice_interface.read_block_vector_data(read_data_id, data_size,
        ↪  precice_ids, read_data)
        # Deform grid based on scatfile in parafile
        deform.run(read_primgrid=1, write_primgrid=1)
        tau_python.tau_parallel_sync()
# Finalize precice and solver environment
precice_interface.finalize()
solv.finalize()
# Cleanup of TAU-Python environment
tau_python.tau('exit')
```

preCICE Python bindings methods into existing code:

1. The preCICE environment is initialized in (lines 14 and 15), where the preCICE `Interface` object is instantiated. The arguments for the `Interface` are used to determine if the simulation is run in serial or parallel. The call to `configure()` then configures the `Interface` object via an external preCICE configuration file and sets up communication between parallel processes inside the solver, if necessary.

2. The coupling interfaces are then defined in lines 24-28. The discretizations of shared interfaces in preCICE are known as *meshes*, each with a specific name defined in the preCICE configuration file. Each coupled solver defines the discretization of their preCICE meshes—in this case, the points in the discretisation are given by `data_coordinates`. Each mesh can contain multiple types of data that share the same discretization. In this case, the code passes the data `data_name`. Each mesh and data type is defined by a unique integer value, `mesh_id` and `data_id`, respectively.

3. Once the various interfaces are defined, the `precice_interface` can then be initialized in line 31. This step sets up the various data structures required for the shared interfaces and the communication channels between coupled solvers.

4. The simulation loop is then started. Note that preCICE now controls the simulation code with the auxiliary method `is_coupling_ongoing()` in line 33. preCICE ensures that the number of coupled iterations required by the coupled simulation is performed before the simulation finalizes.

5. In a tightly-coupled simulation, data is exchanged with each simulation time step This is triggered by lines 41-45. The preCICE method `advance()` accepts the time step of the current solver and acts as a synchronization barrier across all coupled participants; the method ensures that solvers with larger time steps effectively "wait" for slower solvers prior to performing coupling steps. Hence, data is written to coupled solvers at the end of each time step and read at the start of each time step (after synchronization) to ensure that data is time accurate between participants. The `read_block_vector_data()` and `write_block_vector_data()` methods read and write data from and to coupled participants. preCICE supports the passing of both scalar and vector data.

6. Once the simulation is concluded, the preCICE environment is finalized (line 50). This performs cleanup to ensure that communication channels are properly destroyed and used memory buffers are freed.

Note that this code listing provides a very bare-bones preCICE implementation to give an idea of how the library can be incorporated into existing code. preCICE additionally supports more complex coupling logic, such as check-pointing, implicit coupling schemes and post-processing of boundary data [47].

5. Adapter Architecture

This chapter focuses on the architecture of the TAU and CAMRAD II adapters, namely the structure of the code and the various techniques employed. The chapter first lists the various goals of the adapter before providing a comparison of the original code and the updated adapters. Finally, the interfaces of the CAMRAD II and TAU adapter are described in greater detail.

5.1 Goals

The main deliverables for this master thesis include the preCICE adapters for TAU and CAMRAD II. As mentioned in the motivation, these adapters must be modular, extensible and easy to use.

In this context, modularity is the ease of switching out solvers. One of the major benefits of preCICE is that once an adapter is created for the solver, the interface to couple said solver with any other previously made adapter is then possible. The adapters written in this work will allow TAU and CAMRAD II to freely couple with existing and future preCICE adapters.

Extensibility refers to the ease of introducing new features or modifying features within existing code. Several features have been proposed by the Chair of Helicopter Technologies, the most pertinent being allowing the modification simulation loop logic, which widely differs between simulations. Additionally, the code should be able to handle changes to the structure of the rotorcraft simulation, the integration techniques for calculating aerodynamic loads and possibly include additional simulation participants.

The final requirement is ease of use. The code is intended to be made available throughout the Chair. Hence, the code must be easy to use and to debug. Note that the main text of this master's thesis only gives a brief overview of how to run adapter code and extensibility. Please refer to Appendixes A, B and C for the user guide, configuration guide and extension guide respectively.

5.2 Workflow of Original Code

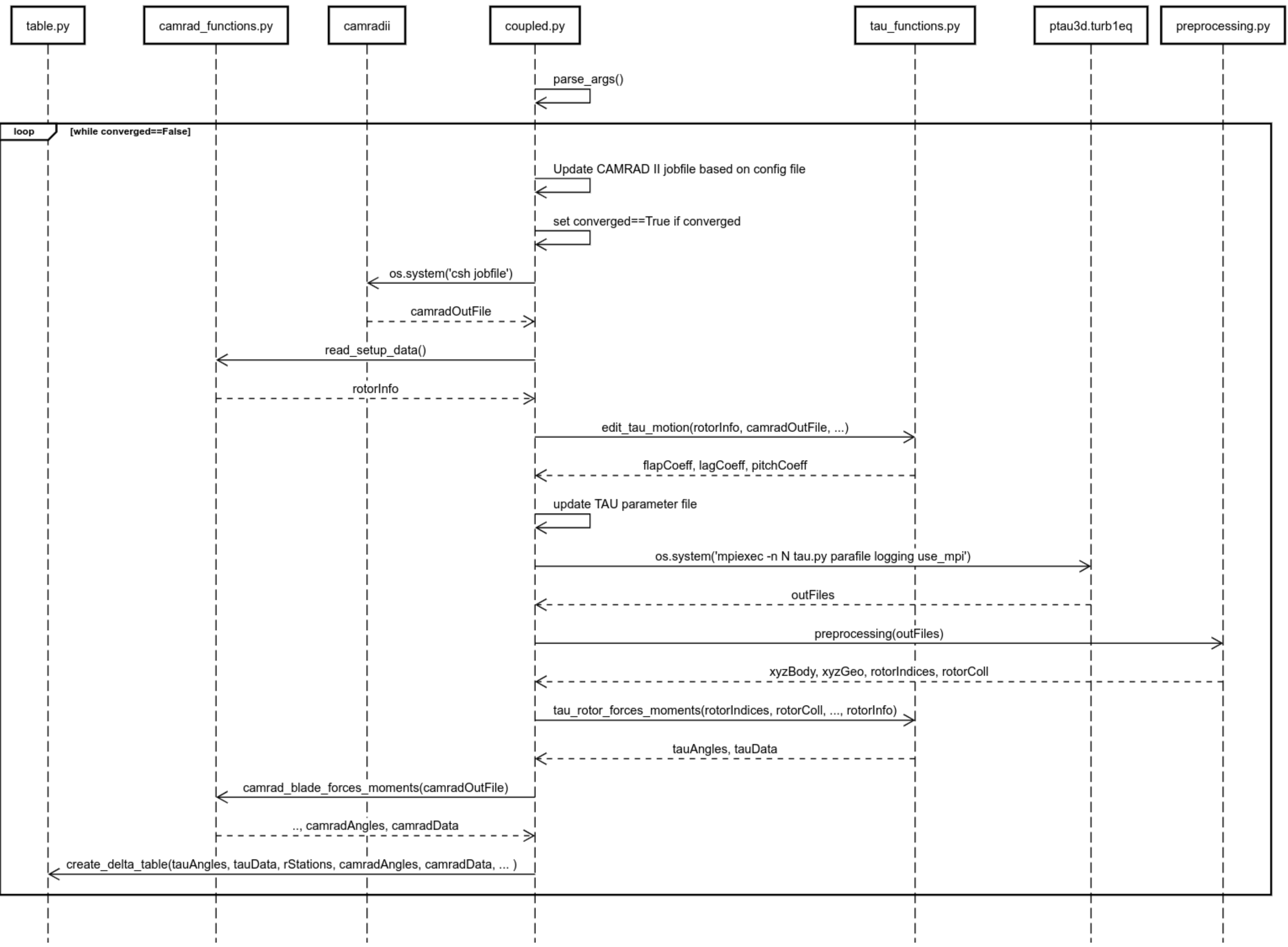
The original coupling implemented at the Chair of Helicopter Technologies resulted in an efficient TAU-CAMRAD II coupled rotor blade simulation. The code was implemented in a single Python script that used methods across multiple files. The general algorithm of the original coupling simulation is shown in Algorithm 5 [1].

Algorithm 5: Coupling algorithm implemented in *coupled.py* [1].

```
1  setup logging using config file
2  read input file and validate parameters
3  write zeros delta table as a placeholder
4  load restart data if available
5  while not converged do
6      create new output directory
7      update CAMRAD job file with new settings
8      if CAMRAD data is missing then
9          run CAMRAD job
10     read control values from CAMRAD output
11     if control value residual < tolerance then
12         set converged = True                    ▷ Simulation converged
13         exit loop
14     read CAMRAD output and update TAU motion file with new blade
       motion
15     update TAU parameter file with new motion file name and output di-
       rectory
16     if TAU data is missing then
17         run TAU
18     if restart data is unavailable then
19         partition mesh
20         save partition data for restarting
21     calculate forces and moments from TAU output data
22     calculate forces and moments from CAMRAD output data
23     save data from TAU and CAMRAD to results file
24     write delta table using the output data
```

The resulting coupling algorithm is clear and easy to understand—a serial-explicit style coupling occurs between CAMRAD II and TAU; CAMRAD II control values are used to determine convergence. However, the underlying code implementation can still be improved. The workflow from the original code is shown in a Sequential Unified Modeling Language (UML) diagram: Fig. 5.1. The folder structure of the original coupling script and simulation inputs is presented in Fig. 5.2 [1].

Figure 5.1: Original Workflow Sequential UML Diagram.



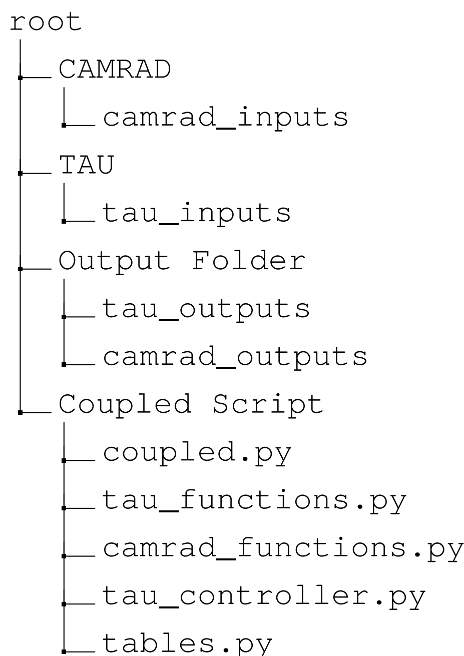


Figure 5.2: Original Coupling Folder Structure.

The first step to reaching the three goals is to ensure the independence or modularity of the adapter code. This means that methods which access input or output data from both solvers, such as `edit_tau_motion()`, must be reformatted so they only access data from a single solver.

Secondly, we can note that the type of data exchanged between the two solvers is fixed in the algorithm—data is exchanged in specific, non-standardized formats. Changing the data exchange or how post-processing works is an involved process that would require a deep understanding of the current code. By creating a standardized API on how data is parsed and handled internally, we can make the code more readable and easier to extend.

We can also note from Fig. 5.1 that there is a high level of coupling and low cohesion in certain parts of the code. For example, the main code `coupled.py` handles a majority of the responsibilities in the algorithm. We can likely simplify this script by refactoring certain functionalities and wrapping them in the right classes.

Finally, we have to ensure that ease of use is maintained. The original code introduced a single configuration file that could control the entire coupling. Similarly, the coupling could be called with a single command. We have to ensure that updating the code interface does not introduce additional complexities of use.

5.3 Workflow with TAU and CAMRAD II Adapters

Based on the remarks in the previous section, we proceed with defining the code structure. A general overview of the two adapters was created and can be seen in Fig. 5.3.

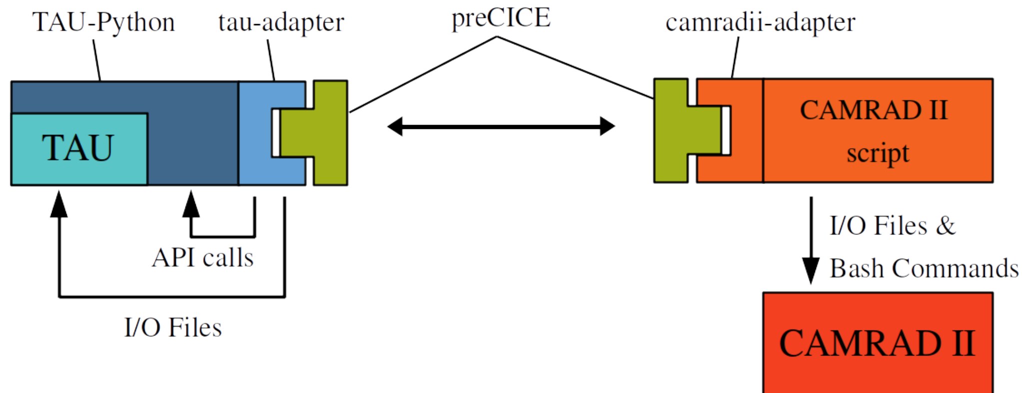


Figure 5.3: Overview of TAU and CAMRAD II Adapters.

The TAU adapter combines calls to the preCICE library and the `TAU-Python` API. The CAMRAD II adapter is a script that runs the CAMRAD II executable via bash. The two adapters modify the input files mentioned in Sections 4.2.2 and 4.1.3. It is worth noting that typical preCICE adapters are not so complex; additional difficulty stems from limitations of the used solvers and the preCICE library for these specific use-cases.

With this overview of the code, we can now look at the code structure of the individual adapters, starting with the more complicated TAU adapter. The TAU adapter code takes an OOP approach and uses a set of classes which represent the core components of a given TAU simulation. These classes can be seen in the UML diagram Fig. 5.4.

The `Adapter` class initializes the simulation environment and the various simulation components, namely the `DataInterface` and `Solver` objects shown in the UML diagram. The `DataInterface` class serves to perform communication with other solvers via preCICE and controls multiple `DataHandler` objects. As the simulation loop and datahandling must be customizable, `Solver` and `DataHandler` are abstract base classes, standardizing their interactions with other classes. The `Solver` base class can be subclassed to implement custom TAU simulation loops, while the `DataHandler` base class can be subclassed to implement new data processing steps.

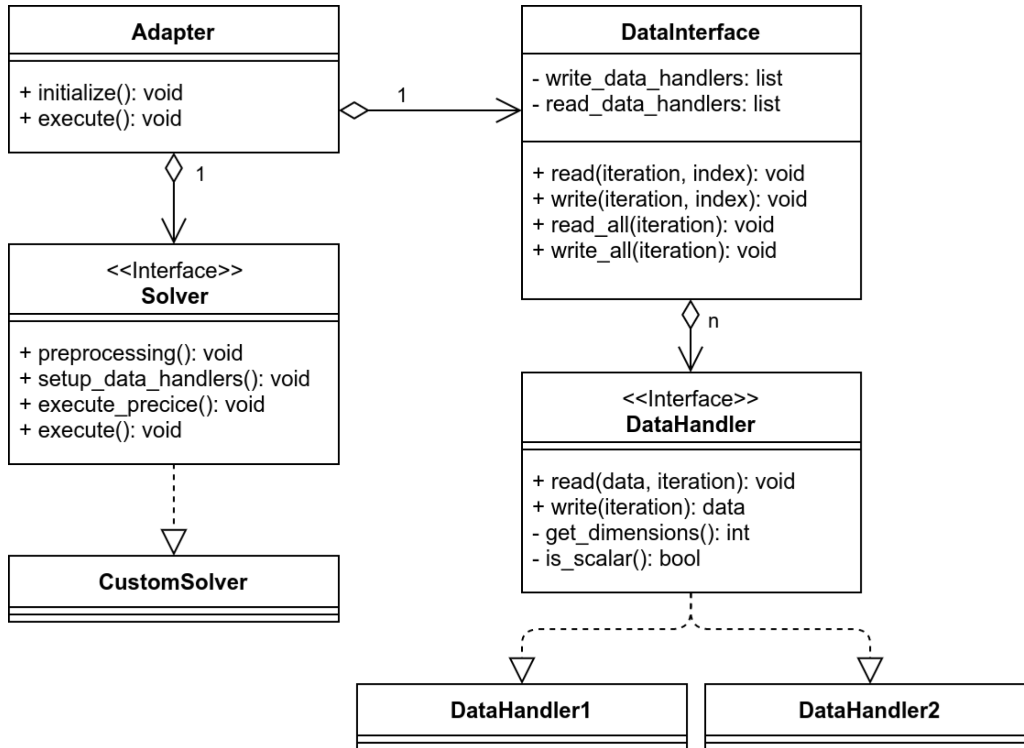


Figure 5.4: Updated TAU UML Diagram.

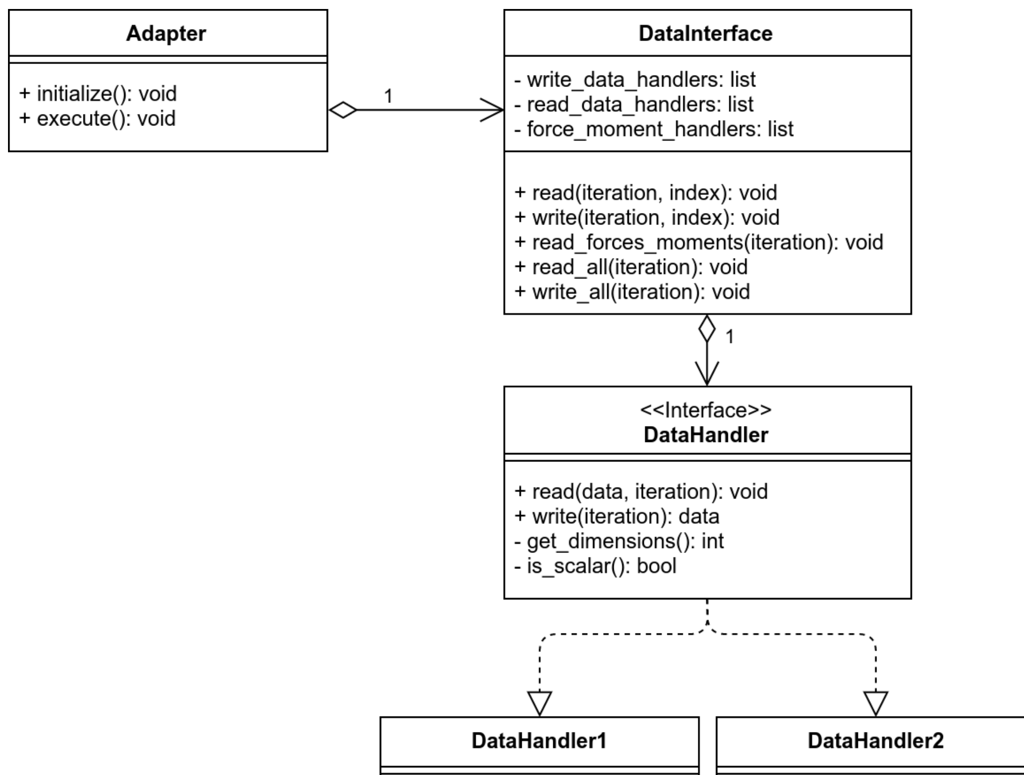


Figure 5.5: Updated CAMRAD II UML Diagram.

The CAMRAD II adapter code does not require the same level of complexity as the TAU adapter. This is because the simulation loop is controlled via input files and we cannot modify the source code. Instead, we create a simplified version of the TAU adapter for CAMRAD II, seen in Fig. 5.5. The `Solver` base class is removed entirely as the simulation loop is sufficiently simple to handle `DataHandler` objects.

Now that we have a basic idea of the various components of the adapters, we can take a look at folder structure. In order to create a simple structure, we separate the base adapter code (which is not modified by the user) and the simulation-specific files into two folders. We refer to the first folder as the *library folder* and the second as the *simulation folder*. This naming is motivated by the fact that the library folder can be converted to a Python library that could be uploaded to PyPI and installed via pip for easy distribution and convenience.

The library folder for the TAU adapter is shown in Fig. 5.6. The classes which apply to every simulation are defined in the top-level folder *tauadapter*. These include the `Adapter` and `DataInterface` classes and the abstract base classes `DataHandler` and `Solver`, all of which do not need to be modified when creating a customized simulation.

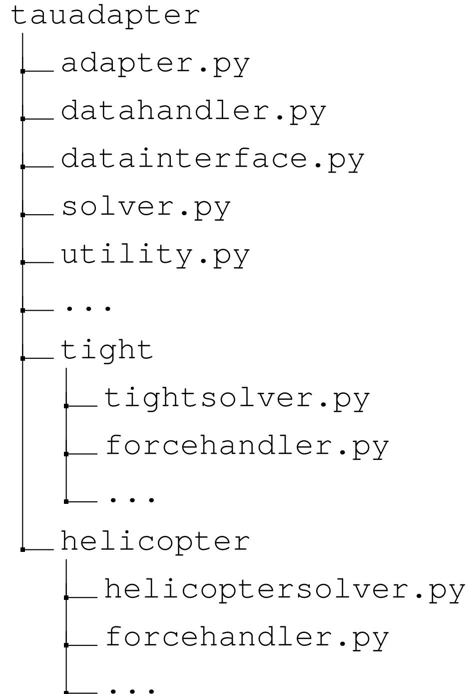


Figure 5.6: TAU Library Folder Structure.

Then, two subclassed `Solver` classes, `TightSolver` and `Helicopter-Solver`, are provided. The two solvers run a tightly-coupled simulation and

a loosely-coupled rotor blade simulation respectively. These are defined in the folders "tight" and "helicopter". The subclassed `DataHandler` classes used by these solvers, such as `ForceHandler` are also included in these folders. The folder structure reflects the class hierarchy of the API and allows for the addition of new verified `Solver` and `DataHandler` subclasses. Each new `Solver` subclass should be placed in its own folder, accompanied with its custom `DataHandler` subclasses. This is to ensure that `DataHandler` subclasses with the same name (such as `ForceHandler`) are not confused when running different types of simulations.

The name "loose" is avoided when naming the `HelicopterSolver` as it is a highly-specialised subclass of `Solver` that is intended only for isolated rotor blade rotorcraft simulation. This is because, in this work, the coordinates of nodes on each rotor blade are aligned to a single reference blade before setting up preCICE mesh vertices and output data is also aligned before being passed to the solid solver (this facilitates the aggregation of multiple timesteps of data when communicating with the solid solver). As a result, the pre- and postprocessing steps of preCICE mesh coordinates and TAU output data are unique for isolated rotor blade simulations and the `HelicopterSolver` cannot be generalized for other loosely-coupled simulations.

In constrast to the library folder, the simulation folder contains the simulation inputs, scripts and user defined subclasses. An example of a TAU simulation folder is seen in Fig. 5.7. The simulation folder contains all the basic components of a TAU simulation, namely the parafile, a primary grid file and an output location. A script to run the coupled simulation `tau_coupled_script.py` and the coupled simulation configuration file `config.yaml` must be provided by the user. Additionally, user defined `DataHandler` and `Solver` subclasses are stored in the "tau_precice" folder. This folder name is arbitrary and can be modified.

The CAMRAD II library folder is quite similar to that of TAU and is shown in Fig. 5.8. However, we see that additional subclassed `DataHandlers` are provided at the top level. This is because CAMRAD II has a standardized output that does not vary. Hence, the `DataHandler` subclasses used in this work should already be applicable to most CAMRAD II simulations, though new `DataHandler` subclasses can still be created by the user. Additionally, the CAMRAD II `Solver` class should not need to be subclassed as CAMRAD II simulation logic is changed by input parameters and the CAMRAD II `Solver` class only wraps the CAMRAD II executable.

```
tauasimulation
├── tau_coupled_script.py
├── tau_parafile
├── config.yaml
├── grid
│   └── primary_grid
├── output
│   └── heli.pval.unsteady...
└── tau_precice
    ├── userdefinedhandler1.py
    └── userdefinedsolver.py
```

Figure 5.7: Sample TAU Simulation Folder Structure.

```
camradadapter
├── adapter.py
├── datainterface.py
├── datahandler.py
├── ...
├── motionhandler.py
└── azimuthhandler.py
```

Figure 5.8: CAMRAD II Library Folder Structure.

```
camradsimulation
├── camrad_coupled_script.py
├── jobs
│   └── jobfile.lnx
├── config.yaml
├── precice-config.xml
├── inputs
│   └── camrad_inputs
├── output
│   └── camout
└── camrad_precice
    └── customhandler1.py
```

Figure 5.9: Sample CAMRAD II Simulation Folder Structure.

A sample CAMRAD II simulation folder is shown in Fig. 5.9. Similar to the TAU simulation folder, the folder contains all the required inputs for a CAMRAD II simulation, namely the CAMRAD II inputs (in the "input" folder) and the CAMRAD II jobfile (in the "jobs" folder). Additional customized `DataHandler` classes are included in the "camrad_precice" folder.

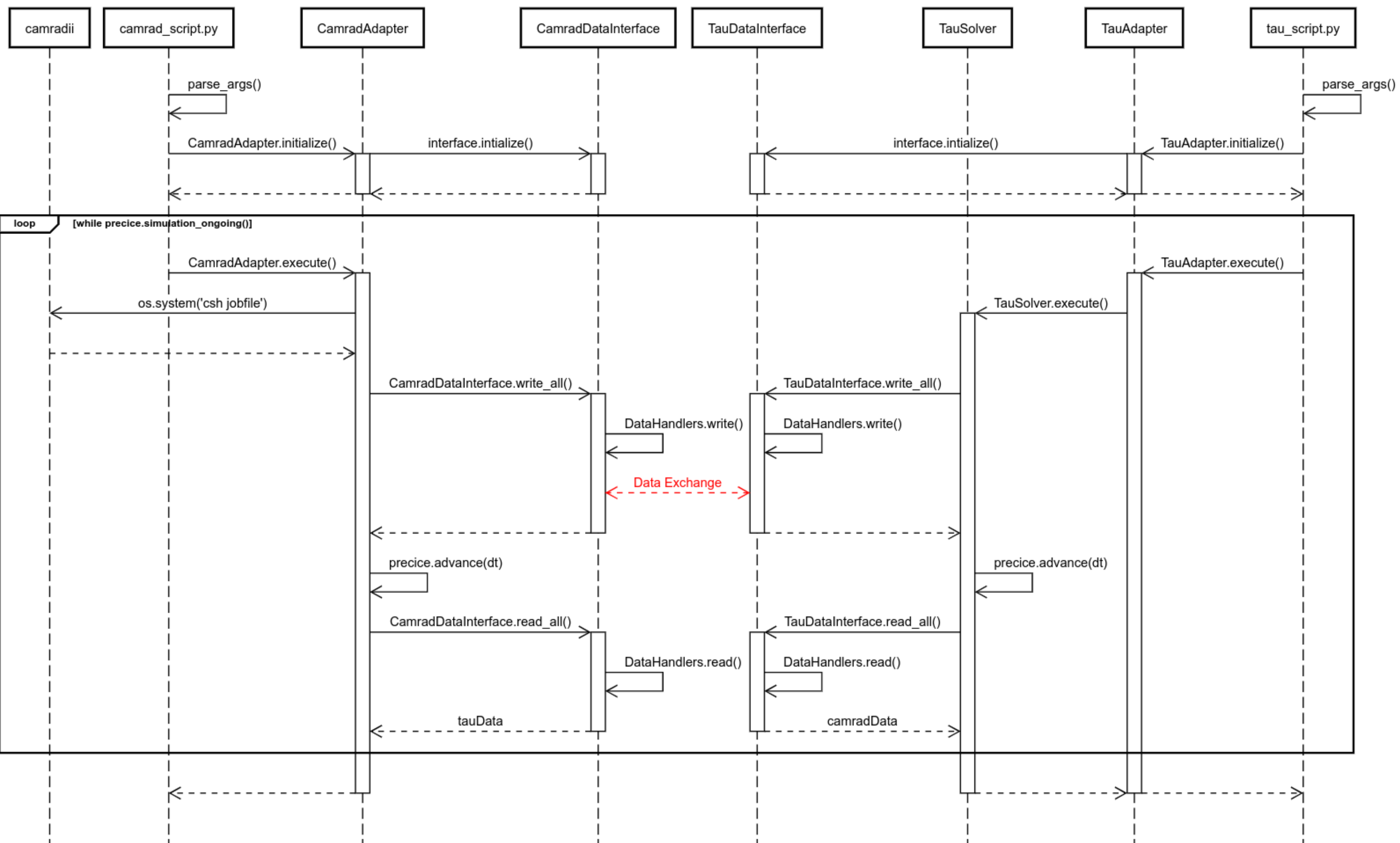
Given this basic code structure, we can now visualize the new workflow in another Sequential UML diagram, shown in Fig. 5.10. We can see from the new sequential workflow that the overall structure of each adapter has been standardized. The responsibilities of each class are more clearly defined. More importantly, the handling of the data is separated from solver simulation loop logic. We can then introduce or remove pre- or post-processing steps in a modular way.

However, the current workflow may seem as complicated or even more complicated than the original workflow but since the interactions between classes has been standardized, a large portion of the code no longer requires attention from the user during runtime. The `Adapter` manages the specific `Solver` and the `DataInterface` handles the individual `DataHandlers` automatically. Hence, from the point of view of the user, the workflow is now simplified to what is seen in Fig. 5.11.

Additionally, we can see from either workflow that we have removed all dependencies, other than when data is exchanged, between the two adapters. Now, interaction between solvers only takes place in the `DataInterface` class (highlighted in red in the Sequential UML diagrams), while major areas of customization of the two solvers are retained. This structure is more modular, flexibile and maintainable.

As the general workflows have been presented, the following two sections will describe the specific API of the TAU and CAMRAD II adapters respectively.

Figure 5.10: New Workflow Sequential UML Diagram.



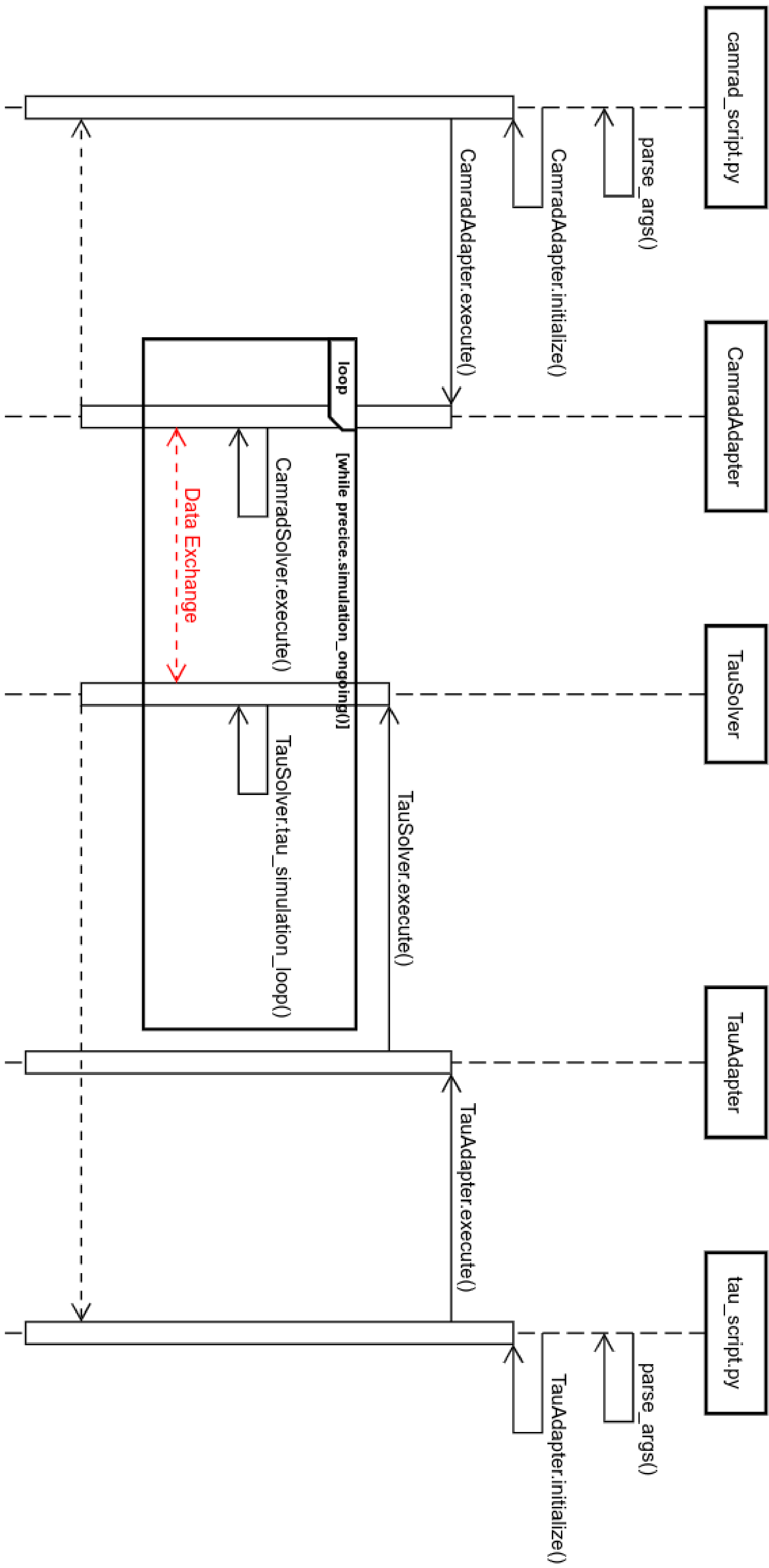


Figure 5.11: Simplified New Workflow Sequential UML Diagram.

5.4 TAU Adapter API

The following subsections will introduce the various classes in the TAU adapter.

5.4.1 Adapter class

The `Adapter` class is the user interface of the `TAU-Python` adapter. This class oversees the initialization and utilization of the preCICE interface and the TAU solver and designed for ease of use. Typical users that do not develop the code should only need to know the `Adapter` API to use the code. A sanitized version of the various `Adapter` class methods can be seen in Listing 5.1.

Listing 5.1: Adapter class for TAU adapter.

```
class Adapter(object):
    """Adapter class controls the DataInterface and Solver classes
    Oversees the instantiation of the two classes and their interactions"""

    def __init__(self, config_file):
            """ Adapter class constructor. Reads configuration file, sets of basic
            ↪  variables """

    def initialize(self):
        """ Main function called to initialize precice environment, TAU environment,
        Solver subclass. Also calls Solver to setup DataHandler objects for
        DataInterface
        """

    def initialize_solver(self):
        """ Determines type of Solver subclass to instantiate based on configuration
         file, run TAU preprocessing if necessary """

    def initialize_precice(self):
        """ Instantiate and configure precice interface via precice.Interface
        class """

    def initialize_tau_python(self):
        """ Initialize tau_python environment (parallel or sequential) by call to
        tau_python.tau_init() """

    def execute(self):
        """ Runs simulation loop in Solver subclass. Also supports running
        simulation loop without preCICE if coupling is not needed """

    def update_tau_solver(self, solver_name, solver_class):
        """ Set custom solver class """

    def update_config(self, config_class):
        """ Set custom config class """

    def add_data_handler(self, handler_name, handler_class):
        """ Add a custom DataHandler subclass """

    def add_data_handlers(self, handler_dict):
        """ Add multiple DataHandler subclasses """
```

Usage of the `Adapter` is simple: initialize the preCICE environment and the `TAU-Python` environment before instantiating the solver by calling the `initialize()` method and setting up data handlers via the `setup_data_handlers()` method. Then, run the solver via `execute()`. Note that the `execute()` method is able to run the solver coupled with preCICE or completely stand-alone (without coupling) depending on the input paramers in the config file.

A typical example of code running the TAU simulation is seen in Listing 5.2.

Listing 5.2: Sample Adapter Code for TAU adapter.

```
1  from tauadapter.adapter import Adapter
2
3  config_file = "config.yml"
4  adapter = Adapter(config_file)
5
6  adapter.initialize()
7  adapter.setup_data_handlers()
8  adapter.execute()
```

Note that additional customizability is also provided. The `update_tau_solver()` method and `update_config()` method allows the addition of a completely new `Solver` or `Config` subclass. Users can also include new `DataHandler` subclasses via the `add_data_handler()` or `add_data_handlers()` methods. An exact guide on how to use these methods is provided in Appendix C.

5.4.2 DataInterface class

The `DataInterface` class serves as the interface between the TAU solver and the other coupling participants. This class controls multiple `DataHandler` objects in the simulation, allowing control of when and what information is passed or received from the various coupling participants during the simulation. By design, The `DataInterface` is the single interface by which the TAU adapter communicates with simulation participants via the preCICE API. The various method signatures in this class are given in Code. 5.3.

The `DataInterface` class has six methods: `add_read_data_hander()`, `add_write_data_hander()`, `read_all()`, `write_all()`, `read()` and `write()`. The first two methods add a specific read or write `DataHandler` to `DataInterface` while the latter four control the passing of information to and from preCICE. The `DataInterface` object supports the read/write of all

Listing 5.3: DataInterface class for TAU adapter.

```
class DataInterface(object):
    """ DataInterface class controls the communication between TAU and preCICE"""

    def __init__(self, precice_interface, config):
        """ DataInterface constructor. Creates empty lists to store DataHandlers
        that read or write simulation data. (self.data_readers and
        self.data_writers respectively) """

    def initialize_data(self):
        """ Function that reads initialized data, to be used if coupling scheme is
        serial """

    def add_read_data_hander(self, data_reader):
        """ Appends DataHandler object to self.data_readers list. Returns index of
        appended DataHandler """

    def add_write_data_hander(self, data_writer):
        """ Appends DataHandler object to self.data_readers list. Returns index of
        appended DataHandler """

    def read_all(self, iteration):
        """ Loops through all existing data readers, communicates with preCICE
        interface and passes data to the DataHandler objects for post-processing """

    def write_all(self, iteration):
        """ Loops through all existing data writers, communicates with preCICE
        interface and passes data from DataHandler objects to preCICE """

    def read(self, iteration, index):
        """ Function that communicates with preCICE and passes data to a specified
        DataHandler object for post-processing """

    def write(self, iteration, index):
        """ Function that communicates with preCICE and passes data from a specific
        DataHandler object to preCICE"""
```

`DataHandler` objects via a single call via `read_all()`, `write_all()`. The methods to add `DataHandler` objects returns the specific index of the added `DataHandler`, which allows the user to read/write using a specific `DataHandler` via the methods `read()` and `write()`.

5.4.3 Config class

Before described the two customizable base classes, the `DataHandler` and `Solver`, it is apt to introduce the `Config` class. A set of classes inherting from the Python jsonobject library is used to read input variables, store runtime data and share information between different components (such as different `DataHandlers`). A sample `Config` class structure is seen in Listing 5.4.

The main configuration file is read as a `Config` object, which stores the input variables required to start TAU and preCICE. These include the name of the TAU or preCICE configuration files, the parafile location, and folder structure

Listing 5.4: Sample Config yaml for TAU adapter.

```
logging_location: ./log
precice: true
preprocessing: false
participant: TAU
parafile: parafile_orig
precice_config: precice-config.xml
restart_location: ./restart
solver: HelicopterSolver

interfaces:
- mesh: TAU_Collocations
  size: 200
  read_data:
  - CollocationsHandler
- mesh: TAU_Mesh0
  boundary_markers: [5, 6, 15]
  blade: 0
  write_data:
  - ForcesHandler_0_0
  - MomentsHandler_0_0

 simulation:
 - total_azimuths: 24
 - interpolation_type: piece-wise
```

information. The contents of the `Config` object should not be changed during runtime of the coupled simulation.

The attribute "interfaces" is a list of `Interface` jsonobjects that define the preCICE meshes and specify specific `DataHandler` subclasses. There is a specific syntax required when defining the "interfaces" attributes and further detail is provided in the Appendix A. There are two preCICE meshes defined in the given example. The first, "TAU_Collocations", passes non mesh-based data (collocation points or CAMRAD II sensor positions). This is indicated by the "size" attribute. The contents of "read_data" indicates the specific `DataHandler` class used to read collocation data. The second preCICE mesh, "TAU_Mesh0" writes aerodynamic force and moment data. These data are mesh-based. This is indicated by the presence of the "boundary_markers" attribute, which is a list of associated TAU boundary markers.

Finally, the "simulation" attribute is read as `Simulation` jsonobject. This class acts as a shared pointed between multiple `DataHandlers` in the case of data dependencies. This exploits the fact that Python effectively uses "pass-by-reference" to allow different objects to share data.

There are several advantages to using the Python jsonobject library. Firstly, it can apply requirements to input variables, such as enforcing type-setting, raising errors if required variables are not set and inherently supports default

values for variables. Secondly, jsonobjects are flexible. New object attributes can be added directly and accessed without prior definition; the `Simulation` "shared-pointer" is easily used. Finally, the `jsonobject.Jsonbject` class is easily subclassed for high customizability (see Appendix C for more details).

In this example, the `Interface` class has been subclassed to also provide the "blade" attribute (seen in Line 17) used to rotate (and align) output data to the correct orientation in a multi-bladed rotorcraft simulation. This allows users to add custom required attributes when reading user defined configuration files.

5.4.4 DataHandler class

The DataHandler class is an abstract class that defines how coupled data is proccesed in the simulation: either data sent from TAU or data received by TAU from other coupling participants via preCICE. There were several problems when designing this class as it had many requirements. Firstly, the base class needs to have a standard interface—so that a standard `DataInterface` class would be able to handle new user defined subclasses instances. Additionally, the fluid and solid solver introduce *non-mesh* based data, such as azimuth positions, Fourier series and collocation points. This data is not associated with a particular mesh or discretisation, which preCICE was not designed to handle. Finally, the standardization should not reduce the flexibility of the code. The function definitions in the base class are shown in Listing 5.5.

To address the first issue, we designate two virtual methods using the Python *ABCMeta* library `read()` and texttwrite(), while trying to keep the syntax as simple as possible. Understanding that preCICE passes either scalar or vector data, we implement the `is_scalar()` virtual method used by the `DataInterface` class to determine which preCICE method to call.

The `read()` method is analogous to the `read_block_scalar_data()` and `read_block_vector_data()` found in the preCICE library. This method accepts data as a 1D array (as provided by the preCICE API) and the coupling iteration number (sometimes required to write output files) and post-processes the received data.

The `write()` method is analogous to the `write_block_scalar_data()` and `write_block_vector_data()`. This method accepts the coupling iteration number, reads the required data from a TAU memory buffer or an output file and returns a 1D array of processed data.

We want most of the calculation steps to be handled by these objects so that the simulation loop can be kept generic; the resultant modularity of `DataHan-`

Listing 5.5: DataHandler class for TAU adapter.

```
class DataHandler(object):
    """ DataHandler class controls the pre- or post-processing of a single data
    type """
    def __init__(self, args):
        """ DataHandler constructor. Sets up self.mesh_id and self.data_id """
        self.mesh_id = args[constant.MESH_ID]
        self.data_id = args[constant.DATA_ID]
        self.precice_vertex_ids = args[constant.PRECICE_VERTICES]
        self.length = len(list(self.precice_vertex_ids))

        self.vertex_ids = args[constant.VERTEX_IDS]
        self.coordinates = args[constant.COORDINATES]

        self.tau_para = args[constant.PARAFILE]
        self.comm = args[constant.COMMS]

        self.config = args[constant.CONFIG]
        self.interface = args[constant.INTERFACE]
        self.simulation = self.config.simulation

    @abstractmethod
    def write(self, iteration):
        """ Abstract function used to write data to preCICE """

    @abstractmethod
    def read(self, data, iteration):
        """ Abstract function used to read data from preCICE """

    @abstractmethod
    def is_scalar(self):
        """Function returns True if data type is scalar """

    @abstractmethod
    def get_dimensions(self):
        """Function returns number of dimensions of simulation as int """
```

`dlers` also allows us to avoid as much unnecessary calculations as possible.

Hence, we have to address the information required by an arbitrary `DataHandler` class to peform pre-processing or post-processing—what data would a user need for most calculation steps. So, we list the data required for communication with preCICE and pre- or post-processing of data.

In order to send or receive data from preCICE, we need four pieces of information: the mesh id, the data id , the preCICE vertex ids and the length of the sent or received data. Then, we need the relevant TAU vertex ids (or the global node ids) to access mesh-based data from TAU. Some calculations required the exact coordinates of the relevant nodes. So, these are provided as well. Some pre- or post- processing requires updating the TAU parafile or reading data from the parafile. Hence, the PyPara object is provided. Finally, the customizable jsonobjects storing configuration, simulation or interface data are provided. These provide users with an interface to pass arbitrary data to the `DataHandler` on the fly. Finally, an mpi4py Intracomms object is passed

that allows communcation between parallel processes.

The base `DataHandler` class now looks quite cluttered with many member attributes. We want this data to be accessible by the user but do not want them to have to keep this process in mind when designing a custom `DataHandler`. Additionally, we do not want users to have to consider if data is mesh-based or not. In order to facilitate this, we contain all the aforementioned data in a single Python `dict`, which is unpacked by the `DataHandler` base class.

A custom class `DataHandlerFactory` was then designed set up `DataHandlers`. This class parses the `config.yaml`, automatically determines the `DataHandler` subclass to instantiate, checks if data is mesh-based or non mesh-based and performs the provides the required data. This wraps several complicated processes, such as determining the node ids associated with a given boundary marker for a given parallel process at runtime and reduces accesses to hard-drive memory as much as possible. This class does not need to be modified by users when creating new `DataHandler` subclasses.

Understanding the overall explanation of this class may be confusing, please refer to Appendix A and C, which provides information on how to subclass and use the `DataHandler` base class.

5.4.5 Solver class

Recalling in Section 4.1, the `TAU-Python` API allows the design of highly customizable fluid solvers which have extremely variable simulation loops. Hence, the replacing of solvers prior to runtime with minimal effort is highly desirable. The `Solver` class is an abstract class (created using the Python *ABCMeta* library) designed to serve as base class for any `TAU-Python` solver. It contains several virtual functions that standardize its interaction with other classes. These methods are shown in Listing. 5.6.

The `Solver` class constructor accepts `config_file` and `data_interface`. These arguments are automatically provided when running the `Adapter` method initialize(). The `update_config()` method is used for a custom `Config` class and `add_data_handler()` allows the inclusion of custom `DataHandler` classes.

When creating a new `Solver` class, the `execute()` and `execute_precice()` methods should be implemented. This allows users to specify the specific simulation loop required. Subclassing the `Solver` method will require some understanding of the preCICE Python bindings. The subclassing procedure will be described in greater detail in Appendix C.

Listing 5.6: Solver class for TAU adapter.

```
class Solver(object):
    def __init__(self, config_file, data_interface, config_class=Config):
        """ Constructor for solver base class.\n
        Instantiates the various tau_python classes required to control the TAU
        simulation"""
        # Instantiate PyPara, PySolv, PyPrep, PyDeform
        self.config = utility.config(config_file, config_class)
        self.data_interface = data_interface
        self.precice_interface = data_interface.precice_interface

        self.datahandlerfactory = DataHandlerFactory(self.config,
                                                     self.precice_interface,
                                                     self.data_interface,
                                                     self.tau_para)
        MPI.COMM_WORLD.barrier()

    def preprocessing(self):
                """ Runs TAU preprocessing if necessary """

    def add_data_handler(self, class_dict, handler_type):
        """ Update dictionary of datahandler types

        Arguments:
            class_dict (dict): dict containing key-value pair, where key is the name
                of the new datahandler and value is the datahandler class type.
            handler_type (str): updates DATA_READERS or DATA_WRITES dict depending
                on input
        """
        self.datahandlerfactory.add_data_handler(class_dict, handler_type)

    def setup_data_handlers(self):
        """ Used to set up the various data handlers defined in config file
        """
        self.datahandlerfactory.initialize()
        for interface in self.config.interfaces:
            self.datahandlerfactory.create_data_handlers(interface)

        self.logger.info("Completed setting up data readers and writers")

    @abstractmethod
    def execute_precice(self):
        """ Abstract method that is unique to each solver case. Executes solver with
        preCICE """

    @abstractmethod
    def execute(self):
        """ Abstract method that is unique to each solver case. Executes solver
        without precice """

    def define_deformation_function(self, function):
        """ Sets the deformation function if run without preCICE """
```

5.5 CAMRAD II Adapter API

The following subsections will introduce the various classes in the CAMRAD II adapter.

5.5.1 Adapter class

The `Adapter` class in CAMRAD II functions as a script for the CAMRAD II executable. It is analogous to the combination of `Solver` and `Adapter` classes in the TAU adapter. The method signatures found in the class are seen in Listing 5.7.

Listing 5.7: Adapter class for CAMRAD II adapter.

```
1  class Adapter(object):
2      def __init__(self, config):
3          """ Constructor for adapter class. """
4
5      @staticmethod
6      def update_config(self):
7          """ Static method used to update Config class in Adapter class. """
8
9      def add_data_handler(self, handler_name, handler_class):
10         """ Add a custom DataHandler subclass """
11
12     def add_data_handlers(self, handler_dict):
13         """ Add multiple DataHandler subclasses """
14
15     def setup_data_handlers(self):
16         """ Sets up data handlers. """
17
18     def initialize(self):
19         """ Executes solver with precice. """
20
21     def execute(self):
22         """ Runs simulation loop in CAMRAD. """
```

The method `setup_data_handlers()` creates the various `DataHandler` objects specified in the configuration file and the method `execute()` runs the simulation loop. No standalone method was written as this can be accomplished by running the jobscript with the CAMRAD II executable directly. Similar to the TAU `Adapter` class, new `DataHandler` or `Config` subclasses can be added with the `add_data_handler()`/`add_data_handlers()` and `update_config()` methods.

Listing 5.8: DataInterface class for CAMRAD II adapter.

```
class DataInterface(object):
    """ DataInterface class controls the communication between CAMRAD and preCICE
    """
    def __init__(self, precice_interface, config):
        """ DataInterface constructor. Creates empty lists to store DataHandlers
        that read or write simulation data. (self.data_readers and
        self.data_writers respectively) """

    def initialize_data(self):
        """ Function that reads initialized data, to be used if coupling scheme is
        serial """

    def add_read_data_hander(self, data_reader):
        """ Appends DataHandler object to self.data_readers list. Returns index of
        appended DataHandler """

    def add_write_data_hander(self, data_writer):
        """ Appends DataHandler object to self.data_readers list. Returns index of
        appended DataHandler """

    def add_force_moment_readers(self, data_writer):
        """ Adds DataHandler object to self.force_moment_readers
        """

    def read_all(self, iteration):
        """ Loops through all existing data readers, communicates with preCICE
        interface and passes data to the DataHandler objects for post-processing """

    def write_all(self, iteration):
        """ Loops through all existing data writers, communicates with preCICE
        interface and passes data from DataHandler objects to preCICE """

    def read_forces_moments(self):
        """ Reads forces moments data from fluid solver and arranges it in order """

    def read(self, iteration, index):
        """ Function that communicates with preCICE and passes data to a specified
        DataHandler object for post-processing """

    def write(self, iteration, index):
        """ Function that communicates with preCICE and passes data from a specific
        DataHandler object to preCICE"""
```

5.5.2 DataInterface class

The `DataInterface` class for the CAMRAD II adapter is similar to that used in TAU. However, there are an additional two methods not present in the TAU `DataInterface` class, `add_forces_moments_handler()` and `read_forces_moments()`. These new methods are required to aggregate force and moment data over a full revolution due to the inherently different outputs of the fluid and solid solvers. CAMRAD II, similar to many contemporary CSD solvers used for rotor blade simulations, calculates data for a full

revolution each time it runs in order to produced a trimmed solution. However, most fluid solvers calculate data for each timestep. Hence, the adapter code must aggregate the multiple timesteps of data passed by the fluid solver when updating CAMRAD II input files.

The passing of periodic data is handled in this work by treating data from each timestep as a new set of data in preCICE. Each set of data is then handled by a single `DataHandler` object in the fluid solver (more details are presented later in Section 6.2.5). The `read_forces_moments()` method aggregates data passed by multiple `DataHandler` objects to generate the aerodynamic loading delta table file required by CAMRAD II.

5.5.3 DataHandler class

The abstract `DataHandler` class for the CAMRAD II adapter are largely identical to that used in TAU. However, unlike TAU, fewer variables are required by the `DataHandler`. Hence, we do not package the various objects into a `dict` object but instead pass them individually. The sanitized `DataHandler` base class is shown in Listing 5.9, with largely similar methods seen in the TAU `DataHandler` class. Refer to Appendix C for more details in subclassing this base class.

Listing 5.9: DataHandler class for CAMRAD II adapter.

```
1   class DataHandler(object):
2       """ DataHandler class controls the pre- or post-processing of a single data
3       type """
4       def __init__(self, mesh_id, data_id, precice_vertex_ids):
5           """ DataHandler constructor. Sets up self.mesh_id and self.data_id """
6           self.mesh_id = mesh_id
7           self.data_id = data_id
8           self.precice_vertex_ids = precice_vertex_ids
9           # Number of dimensions in problem
10          self.length = len(self.precice_vertex_ids)
11
12      @abstractmethod
13      def write(self, iteration):
14          """ Abstract function used to write data to preCICE """
15
16      @abstractmethod
17      def read(self, data, iteration):
18          """ Abstract function used to read data from preCICE """
19
20      @abstractmethod
21      def is_scalar(self):
22          """Function returns True if data type is scalar """
23
24      @abstractmethod
25      def get_dimensions(self):
26          """Function returns number of dimensions of simulation as int """
```

5.5.4 Config class

The CAMRAD II `Config` class is largely similar to that for TAU. The main difference is the removal of boundary markers as an attribute for the `Interface` class and the inclusion of the `RotorInfo` jsonobject, which contains rotor blade definition information. For more details of the inputs for the CAMRAD II configuration files and how to extend the class, please refer to Appendix B and C.

6. Implementation

This chapter takes a closer look at the actual implementation of the TAU and CAMRAD II code for the three test cases implemented in this work: a tightly-coupled perpendicular flap simulation [2], a loosely-coupled case, based on the simulations performed in the original work [1] and a loosely-coupled case that includes deformation of the TAU grid. Recalling the folder structures discussed in Chapter 5, we divide the library folders and the simulation folders into different gitlab respositories. The library folders for the TAU[1] and CAMRAD II[2] adapters are each a gitlab repository. The various tutorials are included in the TAU adapter tutorials gitlab repository[3].

6.1 Toy Example: Perpendicular Flap

A tightly-coupled case was used when designing the basic TAU adapter [2][4]. This case was chosen for its low computation cost compared to the extremely computationally expensive rotor blade simulations. Originally, this tutorial coupled the `OpenFOAM` fluid solver with the `CalculiX` solid solver. The `OpenFOAM` adapter was replaced by the TAU adapter.

Note that, due to time constraints, this coupled simulation was not tuned to match the original simulation. Additionally, the original simulation was itself not designed to produce physical results. Thus, the behavior of the TAU-`CalculiX` coupling is not intended to fully replicate the results from the original work and the results from this simple coupling case are not used to validate this work. Instead, this test case was used in the initial development of the TAU adapter before implementing more complex models.

[1] `https://gitlab.lrz.de/KeefeHuang/tauadapter`

[2] `https://gitlab.lrz.de/KeefeHuang/camradadapter`

[3] `https://gitlab.lrz.de/KeefeHuang/tau-adapter-tutorials.`

[4] `https://github.com/precice/tutorials`

6.1.1 Simulation Setup

This example models fluid flowing through a channel that interacts with a solid, elastic perpendicular flap that is fixed to the channel floor. The setup can be seen in Fig. 6.1.

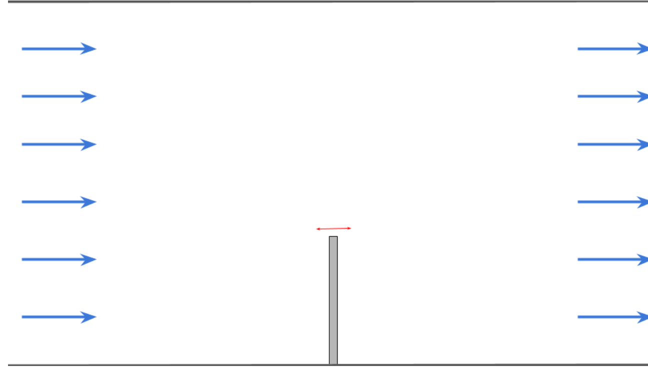


Figure 6.1: Toy Example: Simulation Setup [2].

The flow domain is 6 units long and 4 units high. The flap is located at the center of the length of the flow domain, has a thickness of 0.1 units, a height of 1 unit and a width (into the direction of the page) of 0.3 units. The flow speed is variable and can be manually changed. For this test case, the default speed of 10 units/s was used.

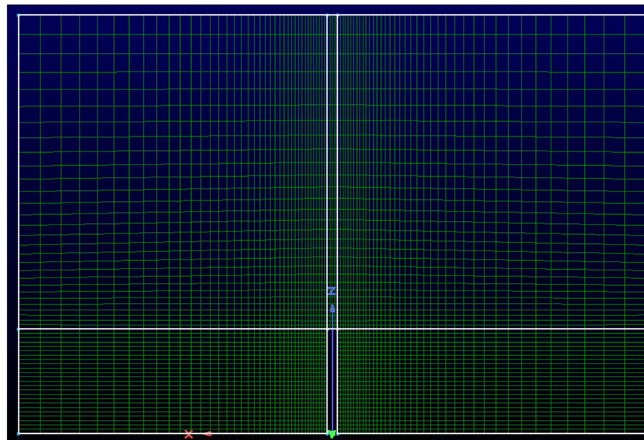


Figure 6.2: Toy Example: Pointwise Grid.

A grid was generated by the pointwise software, shown in Fig. 6.2. During coupling, TAU passes force data at the interface to `CalculiX` and `CalculiX` passes displacement information to TAU.

6.1.2 TAU Solver

A simple TAU simulation loop was written using the turbulent one equation model via the `TAU-Python` API. This simulation loop is implemented in the `TightSolver` class available in the TAU adapter gitlab repository in the `tight` folder.

Algorithm 6: `TAU-Python` algorithm for tightly-coupled case

1 **init** TAU-Python classes: PyPara, PySolv, PyPrep, PyDeform with TAU Parafile
2 **init** solver via PySolv
3 **while** self.precice_interface.is_coupling_ongoing() **do**
4   **init** outer loop
5   **advance** time and motion in TAU simulation
6   **solve** multigrid
7   **run** inner loop n_{in} times ▷ n_{in} = number of inner time steps in Parafile
8   **print** monitoring data
9   **save** output data per Parafile settings
10   **process** and **send** TAU data to solid simulation via DataInterface
11   **update** time in preCICE coupling
12   **receive** and **process** solid simulation data
13   **write** deformation scatfile and call PyDeform class to deform grid
14 **stop** and **finalize** solver
15 **stop** and **finalize** preCICE coupling interface

The coupling runs in serial, with the fluid solver (TAU) running first. Data is passed from TAU to `CalculiX` at the end of each fluid solver time step, seen in Line 10 and data is received from `CalculiX` via preCICE at the beginning of each new solver time step, seen in Line 12.

6.1.3 TAU DataHandlers

In this coupling simulation, the data from the TAU simulation was extracted directly from the TAU output files and no processing of force data was required. Two `DataHandler` subclasses were created: the `ForceHandler` class reads output data from TAU and the `DisplacementHandler` reads data from `CalculiX`. These two classes can be found in TAU adapter gitlab respository.

6.1.4 Folder Structure & Running Tutorial Case

The folder structure for this simulation is slightly different from the simulation folder structures introduced in Chapter 6. The `CalculiX` simulation runs in

the same folder as the TAU simulation, which follows the folder structure of the original tutorial case [2], as seen in Fig. 6.3.

```
flapsimulation
 ├─ precice-config.xml
 ├─ precice-config_parallel.xml
 ├─ tau_coupled_script.py
 ├─ tau_parafile
 ├─ tau_config.yaml
 ├─ config.yml
 ├─ flap.grid
 ├─ flap.bmap
 ├─ output
 │   ├─ heli.pval.unsteady
 │   └─ ...
 ├─ grid_deform
 │   ├─ deformed_grid_0
 │   └─ ...
 ├─ scat_files
 │   ├─ test.scat
 │   ├─ ...
 │   └─ rot_test_0.scat
 ├─ runSolid
 └─ Solid
     ├─ flap.inp
     └─ ...
```

Figure 6.3: Perpendicular Simulation Folder Structure.

To run the simulation, call the TAU adapter code with `tau_coupled_script` with `tau_config.yaml` and run the `runSolid` to run the coupled `CalculiX` simulation.

6.2 Case 1: Loose Coupling without Deformation

After succesfully coupling the TAU solver with an existing `CalculiX` preCICE implementation, the next step would be to couple TAU with CAMRAD II based off an existing coupled simulation developed by Carnefix [1]. This section details the individual simulation setups of the two solvers, the overall coupling workflow as well as the individual algorithms used by each solver.

6.2.1 Simulation Setup

The simulation uses a rotor model based off the rotor tested in the second HART. This was an international joint project by DLR, Office National d'Etudes et de Recherches Aérospatiales (ONERA), National Aeronautics and Space Administration (NASA) Langley, US Army US Army Aeroflightdynamics Directorate (AFDD) and Deutsch-Niederländische Windkanäle (DNW) to better understand rotor BVI with the intent of improving rotor noise and vibration [48]. The test was performed in the 4-bladed Bo-105 main rotor, of which a TAU and CAMRAD II model was been developed [48, 1]. The simulation inputs were set to the baseline test conditions in the HART II test, seen in Table 6.1 [49]. One modification to these parameters was performed in the Carnefix's coupling simulation to reduce time to solution, in which the rotor pitch angle was pitched towards the front by 4.5° instead of the rear [1]. The forwards tilt of the blade would reduce vortices on the blade surface, accelerating the convergence of the simuation.

CAMRAD II Simulation Setup

Recalling the requirements in Section 4.2.1, each CAMRAD II simulation has three necessary pieces: the physical model, the simulation environment, and the simulation logic.

The physical model is defined based on the rotor used in the HART II experiment and can be separated into the structural definition and the aerodynamics definition. The structural definition of the rotor is defined in *nodes* and *sections*. These regions are defined using normalized radial positions along the length of the rotor blade r, where $r = \frac{x_{pos}}{R}$, x_{pos} being the position along the blade and R the blade radius. The nodes, located at $r = 0.22$ and $r = 0.61$, determine the degrees of freedom in the rotor blade. The sections determine the structural properties of the rotor blade, such as chord-wise center of gravity, section mass, and bending stiffness. The rotor blade used in this work is divided into 10

sections, which is represented in Fig. 6.4.

Table 6.1: Rotor geometry and test conditions [49].

Parameter	Symbol	Value	Units
Advance ratio	μ	0.15	-
Air density	ρ	1.2055	kg/m^3
Air temperature	T_∞	17.3	°C
Airfoil	-	NACA 23012	-
Blade chord	c	0.121	m
Blade twist	Θ_{tw}	-8	deg/R
Number of blades	N_{b}	4	-
Pitch moment (pos. forward)	M_y	-20	Nm
Roll moment (pos. right)	M_x	-20	Nm
Rotational speed	ω	109	rad/s
Rotor radius	R	2.0	m
Rotor solidity	σ	0.077	-
Rotor shaft pitch angle (pos. aft)	θ_{S}	-4.5	deg
Rotor thrust	T	3300	N
Zero twist radial location	r_{tw}	0.75	-

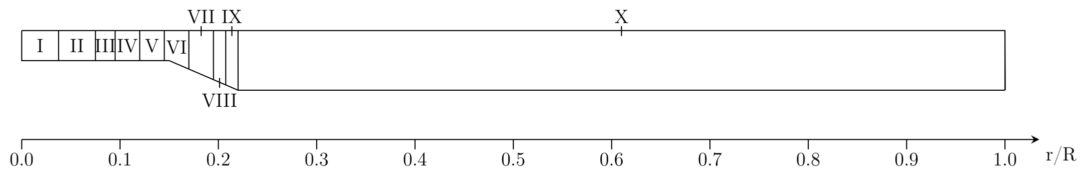


Figure 6.4: Locations of Sections defining the Structural Properties of Rotor Blades in CAMRAD II [1].

The aerodynamics definition of the rotor blade determine where aerodynamic forces and moments can be input or read. The root cutout, sections I-IX, is not relevant to the coupling and is ignored. The remaining portion of the rotor blade is separated into 21 panels, as seen in Fig. 6.5. The 21 panel discretisation is based off studies performed on the CAMRAD II model in [50, 51].

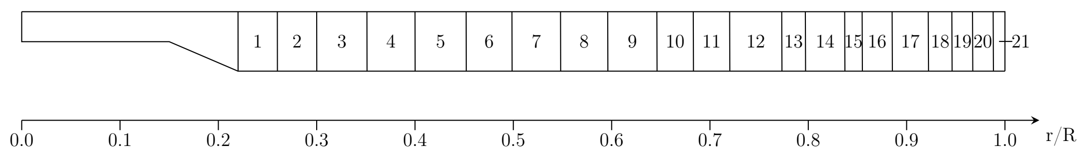


Figure 6.5: Locations of Aerodynamic Panels in CAMRAD II Rotor Blade Model [1].

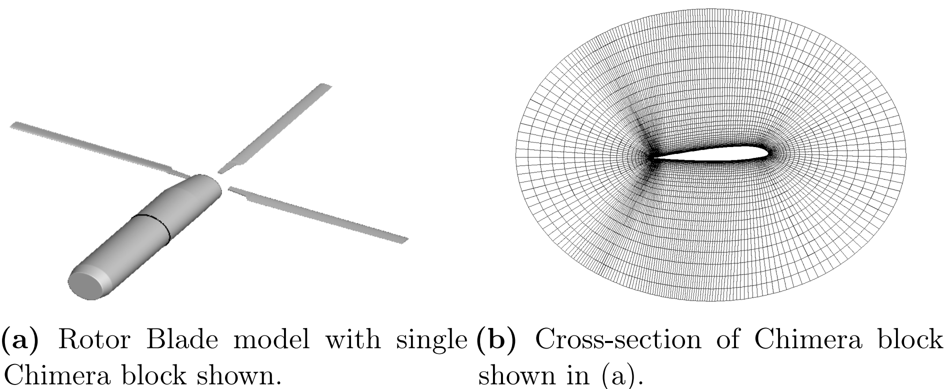
The center of each panel is known as a *collocation point* and determines where motion data is recorded. Collocation points are also the centroids of each panel

and are where the aerodynamic forces and moments are applied when calculated via the internal lifting-line model or supplied by an external Fluid Solver.

The simuation environment mimics that of the HART II experimental setup and the rotorcraft speed is set to zero with a variable wind speed. The simulation runs iteratively to determine a trim solution, using moments as the trim condition. The settings described above are found in the `inputs/HART_II.list` file in the CAMRAD II simulation folder of the tutorial case.

TAU Simulation Setup

The TAU simulation requires a mesh as well as a parafile. For this case, a reference velocity of 34 m/s was used. A cartesian mesh was created using the pointwise software based on the parameters shown in Table 6.1. A chimera meshing strategy, described in Section 3.3, was employed. The chimera block can be seen in Fig. 6.6a and the cross-section of the chimera grid along the shown slice is seen in Fig. 6.6b.



(a) Rotor Blade model with single Chimera block shown.

(b) Cross-section of Chimera block shown in (a).

Figure 6.6: Rotor Blade model for TAU.

The discretisation was performed on a single blade that was rotated multiple times to recreate the 4-bladed model used in the HART II experiments. The TAU simulation runs via the standard Spalart-Allmaras Negative model, which is more robust than the standard Spalart-Allmaras model. In order to achieve sufficiently accurate results for convergence, 360 time steps per revolution were used (effectively one degree per time step). A total of seven full revolutions were run in TAU prior to coupling. This removes non-physical vortices around the rotor blades, increasing the stability of the coupled simulation. 250 inner iterations per time step were used for the initial revolutions, this was reduced to 100 inner iterations during the coupling to speed up the time to solution.

Surface values required for coupling are output every 15°, resulting in 24 output files per revolution. This represents the effective temporal discretisation

of the TAU output data which will be referred to as *azimuth* data for the purposes of coupling. This simulation setup is the same setup described by Carnefix in his initial implementation of this coupling [1].

Note that this simulation was not designed to produce physically accurate results but rather produce results that would result in convergence as a proof-of-concept. For example, an unoptimized mesh is used for simulations. Additionally, this simulation does not consider the elastic motion of the blade, only rigid body motion. Finally, the rotor hub was used as the center of rotation when applying the rigid body motion, which does not accurately reflect the physical behavior of rotor blades.

6.2.2 Simulation Coupling Workflow

These two independent solvers are coupled using a workflow similar to that employed by Potsdam in [18], also known as the delta airloads method mentioned in the literature review. This workflow is most easily described via Fig. 6.7.

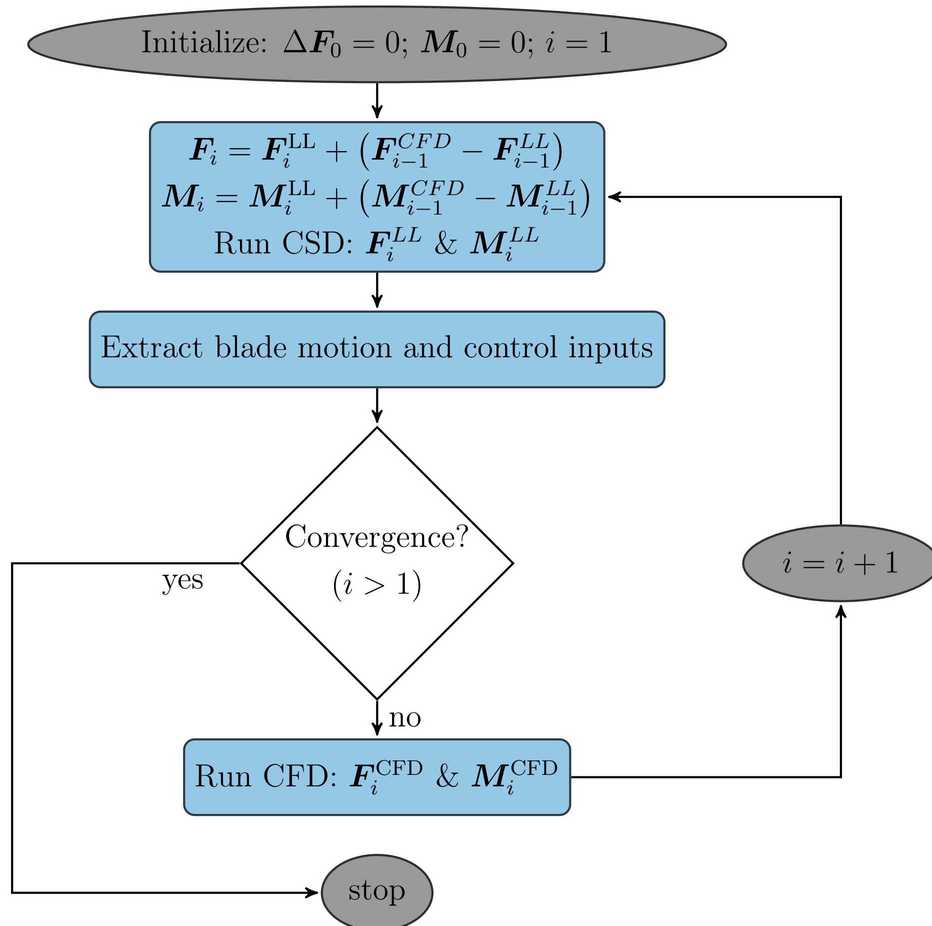


Figure 6.7: Modified Delta Airloads Algorithm Workflow [18].

The simulation is started with CSD simulation. For the initial CSD computation, no CFD data is available. Hence, the simulation runs using the aerodynamic lifting-line method described in Section 4.2.1, producing a trimmed solution F_0^{LL} and M_0^{LL}. Blade motion is passed to the CFD simulation, which computes the aerodynamic loading data F_0^{CFD} and M_0^{CFD}. A correction to the lifting line aerodynamic loads is calculated and the CSD simulation is restarted, producing a new trimmed solution. After the initial coupling step, a convergence check is implemented at the completion of the CSD solver. Unlike the Potsdam example [18], however, which checks both input controls and aerodynamic loads, we provide a simpler convergence criteria; the simulation is assumed to have converged if the trimmed controls converge with a given absolute tolerance, as shown in Algorithm 7. Note that due to the data dependencies inherent in this workflow, the coupled simulation uses a serial-explicit coupling scheme detailed in Section 4.4.2.

6.2.3 CAMRAD II Solver

This coupling scheme means that CAMRAD II is run first, followed by TAU in each coupling step. Recalling that CAMRAD II is a stand-alone executable that produces trimmed solutions. No additional work is required to create a CAMRAD II simulation loop other than updating the aerodynamic loads in the CSD simulation inputs. Otherwise, the CAMRAD II simulation functions as follows in Algorithm 7.

Algorithm 7: CAMRAD II algorithm for loosely-coupled case

```
1  exchange initial data with solid solver
2  while self.precice_interface.is_coupling_ongoing() do
3      run CAMRAD II jobfile, calculate controls C_i        ▷ i is the current
   loose-coupling iteration
4      if ΔC_i = |C_i − C_{i−1}| < Tolerance then
5          signal preCICE that coupling is complete
6      process and send motion and sensor position data to fluid simulation
7      update time in preCICE coupling
8      receive and process aerodynamic load data and time azimuth data from
   fluid simulation
9      update input controls and aerodynamics input table
10 stop and finalize solver
11 stop and finalize preCICE coupling interface
```

Of note in the algorithm is the data that needs to be exchanged between the coupled solvers, seen in Lines 6 and 8. The CAMRAD II solver sends the updated

motion and collocation point data and receives the aerodynamic loading and the azimuth data. Collocation and azimuth data are the additional discretisation information required to interpret the motion data and aerodynamic loading data, respectively.

Unlike the TAU solver case, we do not have an abstract class for CAMRAD II simulation loops. This is due to the fact that we are unable to interact with the CAMRAD II executable. Hence, we have a generic loop that runs CAMRAD II executable. This script is applicable to almost all CAMRAD II simulation cases—as the simulation logic and inputs are defined externally in the input tables, as described in Section 4.2.1. One highly requested feature, error handling of the CAMRAD II scripts, was also implemented. Noting that the coupled script is expected to run for several days and CAMRAD II is run multiple times. This is problematic if the the CAMRAD II script exitted with errors or was suddenly interrupted. The implemented error handling detects and logs errors in the CAMRAD II output and restarts the CSD simulations as necessary.

6.2.4 TAU Solver

The TAU algorithm for the loosely-coupled simulation is seen in Algorithm 8. The simulation loop algorithm for the TAU case is quite similar to that shown in Algorithm 6.

There are four main differences between the two algorithms. Firstly, the initial simulation iterations required to remove the non-physical vortices around the rotor blades prior to coupling are run in Line 3.

Secondly, an additional receive step is implemented in Line 9. We run the two solvers in serial with the CSD solver first to ensure that the CFD solver receives updated information in each iteration; the fluid solver is far more computationally expensive compared to the solid solver and savings from running the CFD fewer times far outweigh the benefits of running the two solvers in parallel. Hence, motion data and sensor position data are read prior to running the CFD solver.

Thirdly, we have to consider that the simulation is now loosely-coupled: data is exchanged periodically, requiring an additional loop inside the coupling loop, seen in 11. This has larger implications on the design of `DataHandler` objects, which will be discussed in Section 6.2.5.

Lastly, for simplicity in this test case, the deformation step has been removed. Referencing the code structure defined in Chapter 5, we create the `Solver` subclass, `HelicopterSolver`, to implement the loosely-coupled simulation loop.

Algorithm 8: `TAU-Python` algorithm for loosely-coupled case

1 **init** TAU-Python classes: PyPara, PySolv, PyPrep, PyDeform with TAU Parafile
2 **init** solver via PySolv
3 **for** $t = 1, 2, \ldots, t_{init}$ **do** ▷ t_{init} = number of initial iterations
4   **init** outer loop
5   **advance** time and motion in TAU simulation
6   **solve** multigrid
7   **run** inner loop n_{in} times ▷ n_{in} = number of inner time steps in Parafile
8 **while** self.precice_interface.is_coupling_ongoing() **do**
9   **receive** *motion* and *sensor position data* from solid solver
10   **update** motion input file
11   **for** $t = 1, 2, \ldots, t_{rev}$ **do** ▷ t_{rev} = number of iterations per coupling step
12   **init** outer loop
13   **advance** time and motion in TAU simulation
14   **solve** multigrid
15   **run** inner loop n_{in} times
16   **print** monitoring data
17   **save** output data per Parafile settings
18   **process** and **send** *aerodynamic loading* and *azimuth* data to solid simulation
19   **update** time in preCICE coupling
20   **receive** and **process** updated motion data from solid solver
21   **update** motion input file
22 **stop** and **finalize** solver
23 **stop** and **finalize** preCICE coupling interface

6.2.5 Data Handling

After the main simulation logic is defined as above, we move on to the actual data handling required by the coupled simulation. Per the code structure defined in Chapter 5, we subclass the `DataHandler` abstract class for each type of data handled. Recalling that the `DataHandler` class can be subclassed to read or write information to and from any coupled solver, this step can be performed generically.

Four sets of data need to be passed: TAU passes aerodynamic data and azimuth data to CAMRAD II, CAMRAD II passes motion data and collocation data to TAU.

Non Mesh-Based Data

preCICE is designed to map data along a shared interface between two arbitrary discretisations. However, for this simulation, there are several handled forms of

data that are not mesh-based and cannot be mapped. Instead, we have to pass exact variables, such as Fourier series coefficients. preCICE has yet to implement this capability, but we can define identical discretizations and use a consistent mapping to pass data between the two solvers—a step which is normally not needed. This might prompt the notion to use another communication protocol instead of preCICE to handle these data formats. However, this is not a trivial undertaking. The new protocol would have to manage communication between multiple solvers, a task which preCICE already accomplishes. This would mean a significant amount of repeated work, which would involve a series of additional verification and validation steps. It is worth noting that the code enhancements required to deal with this problem, such as allowing the re-initialization of preCICE buffer sizes after passing data or handling non-mesh data, are known issues in the preCICE github repository and will be addressed in the future. Hence, the solutions raised in this section are mainly stop-gap measures until the preCICE library supports this functionality in the future.

However, we are currently still faced with this problem—how do we define CAMRAD II buffer sizes? We could manually calculate the buffer sizes apriori. However, this imposes additional responsibility on the user that the configuration files for both TAU and CAMRAD II are updated each time the discretisation is modified. Additionally, the calculation may not be trivial and introduces human error to the situation.

We could also use "sufficiently large" buffer sizes and append additional structural information when passing data. However, the definition of "sufficiently large" depends on which data is being passed. This method works well if we can easily provide a range of buffer sizes apriori. This is the case when passing azimuth data, which describes the temporal discretisation of the TAU simulation output. If we output CFD simulation data every 15°, we define a structure $[25, 0, 15, 30, 45, ..., 360]$, where we store the number of azimuths postions, 25, at the first position in the buffer. Because CAMRAD II only accepts aerodynamic load data at a discretisation of 15°, this limits the buffer size to 25 integers. The user could change the output discretisation in the TAU parafile without consequence. This method also works passingly for collocation data, which requires a maximum buffer size of 2001 floats as CAMRAD II supports up to 2000 sensors per rotor blade. However, this may cause issues when handling motion data.

CAMRAD II provides the absolute displacement at each sensor position in the form of displacement in the x-, y-, z-axis as well as a pitch angle. These

CAMRAD II Output File Excerpt

```
OUTPUT = ROTOR 1 BLADE 4 POS 1.0000R
...
MOTION VALUE            FLAP            LAG             PITCH           AXIAL
...
HARMONICS (FROM 24 TIME STEPS)
  MEAN                  0.341838E-01   -0.392337E-02     2.95509         0.979348
  COSINE   1           -0.153296E-04   -0.482019E-02     2.70720        -0.201911E-05
  SINE     1            0.149269E-03   -0.235455E-02    -1.70087        -0.412899E-04
```

Figure 6.8: CAMRAD II Output File Excerpt.

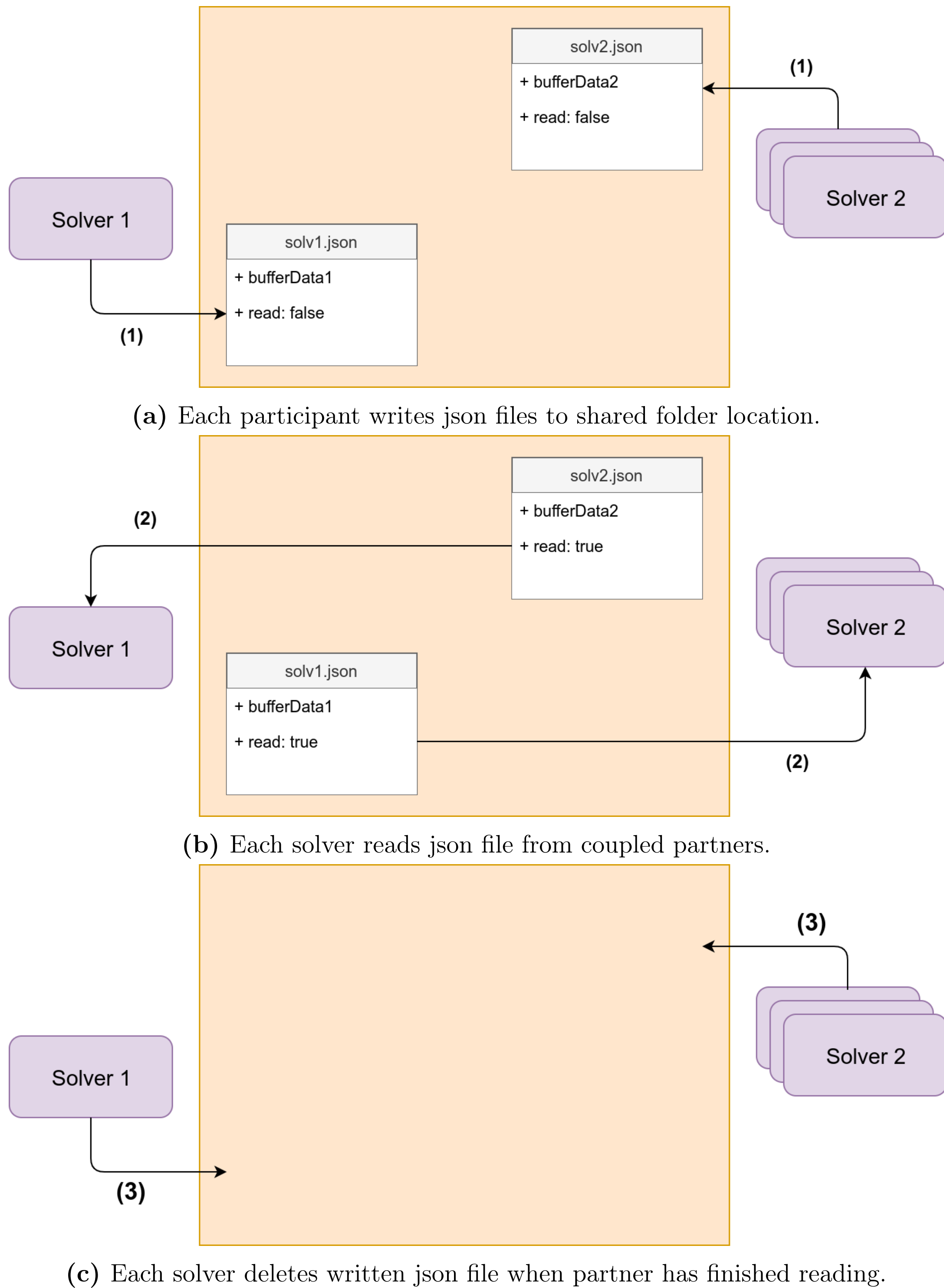
displacements include both rigid body motion and the elastic deformation of the blade. They are referred to as *flap*, *lag*, *axial* as well as *pitch* motions, respectively, and are provided as Fourier series coefficients. An example of the motion data on Blade 4 at the collocation point located at the blade tip or $1.000R$ is seen in Fig. 6.8.

As seen from Fig. 6.8, we need to store twelve floats for motion data at a single collocation point for a single harmonic and CAMRAD II stores up to ten harmonics. Hence, the buffer size for motion data can vary in size immensely. We cannot define a standardized buffer size for motion data and an alternative solution is required.

An ideal solution would be to pass buffer size data between the two solvers prior to defining preCICE meshes. However, this poses a problem as preCICE mesh discretizations must be fully defined before the preCICE `Interface` is initialized. Hence, we cannot use preCICE to pass buffer size data. The feature to pass variable data prior to initializing the preCICE `Interface` is a requested preCICE functionality but development of the feature will only take place after the conclusion of the master thesis. Hence, an additional class was designed to implement this feature, the `Variables` class.

preCICE utilizes a series of written files to define the addresses of each simulation participant. These written addresses are used for inter-solver communication. As a result, a shared directory must be defined in the precice configuration xml. The `Variables` class leverages this existing inherent structure in coupled preCICE code and shares information via files written to the shared directory. The procedure is shown in Fig. 6.9.

Firstly, a json file is written to the shared folder containing the buffer size data, a boolean `read` set to false. Each solver reads the json file from its counterpart, shown in Fig. 6.9b. Then, once all parallel processes have read from



(a) Each participant writes json files to shared folder location.

(b) Each solver reads json file from coupled partners.

(c) Each solver deletes written json file when partner has finished reading.

Figure 6.9: (a) Writing of json file to shared folder location (b) Reading json file by each simulation participant (c) Clean up of read json files.

the json file, the boolean `read` is set to True. This triggers each solver to delete the json file and proceed with the simulation, seen in Fig. 6.9c.

This rudimentary handshaking currently only supports two participants,

though it can be easily extended to multi-solver couplings. The main benefit of such data sharing would be to minimize information overlap between input configurations. Solvers should be able to transmit input variable information such that the user does not need to verify agreement of configuration files prior to startup.

Now that we have a handshaking procedure in place to pass information, we can discuss the actual post-processing of the passed data. The azimuth data from TAU and the collocation point data from CAMRAD II do not require any post-processing and are instead stored in a `Simulation` jsonobject (referenced as needed during runtime). The motion data, however, does require post-processing.

The previously described flap, lag and axial motion data in CAMRAD II output files are provided as displacements. However, the TAU motion file only accepts Fourier coefficients for angle displacements. Hence, the coefficients are approximated as angles:

$$\text{coeff}_\theta \approx arctan(\text{coeff}_\text{displacement}) \tag{6.1}$$

In this simulation, the rigid body motion is approximated by the Fourier coefficients read at the CAMRAD II sensor position the furthest away from the rotor hub, i.e. the far tip of the rotor blades. These coefficients are then used to fill in the TAU motion file described in Section 4.2.1. Of note is that the TAU `PySolv` class is bugged and solver ouput does not reflect the updated motion values despite the solver logging output indicating corrected motion angle values. Instead, the `PySolv` memory must be freed each time motion data is updated to ensure that blade motion is correctly updated.

Mesh Based Data

Having dealt with the non mesh-based data, we can now look at the mesh-based data, which requires a slightly more involved implementation. The force and moment data or the aerodynamic loading data represent the most complex portion of data handling, with multiple difficulties in the coupling. We can split the implementation into two portions: mapping between blades and the passing of revolution data.

The first step is to be able to map aerodynamic loading data from TAU to CAMRAD II. However, this poses some difficulty as TAU has a 3D discretisation, while, as described in Section 6.2.3 and shown in Fig. 6.5, CAMRAD II splits the rotor blade into panels in a 1D discretisation. In order to determine the aerodynamic loading on each panel in the CAMRAD II discretisation, we must aggregate the data in the TAU discretisation in a conservative manner.

As mentioned in Section 4.4.1, preCICE supports nearest-neighbor, nearest-projection and RBF interpolation. However, none are suitable to be directly applied to this 3D-1D mapping as these mappings do not consider the positions of panel edges. This is also a known issue in preCICE and these conversions are typically handled by adapter code. We can see from Fig. 6.5 that panel lengths are irregular as a finer discretisation is used at regions of interest. Hence, if any of the mapping methods are used directly, the cells in the TAU discretisation would not be mapped to the correct collocation point in TAU; Fig. 6.10 demonstrates the result of directly applying the nearest neighbour approach. The figure shows a top-down view of a TAU 3D discretization (at the top of the figure) and a 1D CAMRAD II discretisation below. The data values of each cell in the TAU discretization are mapped to the nearest collocation point of the same color below. Ideally, the red and blue divisions in the TAU discretization should match with the panel edges seen in the CAMRAD II discretization. However, this is not the case in Fig. 6.10; the blue cells highlighted in orange are erroneously assigned to the first panel as they are physically closer to the collocation point of the first panel.

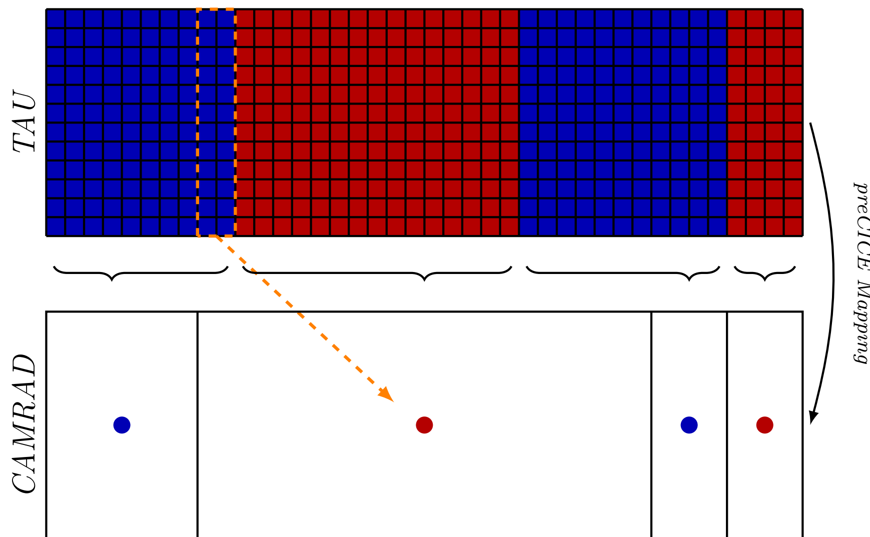


Figure 6.10: Incorrect Passing of Aerodynamic Loading Data from TAU (above) to CAMRAD II (below). TAU cells are allocated to the CAMRAD II collocation point of the same color using a nearest-neighbor mapping. TAU cells highlighted in orange are incorrectly mapped to the first CAMRAD II panel and should be allocated to the second panel.

In order to address this issue, a 2D intermediate grid based on the CAMRAD II discretisation is generated in the CAMRAD II solver code, seen in 6.11. Thus, the preCICE mapping can work as intended when coupling data. An additional

internal aggregation step is then performed in the CAMRAD II code to compute the aerodynamic loads at each collocation point. The intermediate grid provides additional structural information required by the preCICE mapping techniques to work.

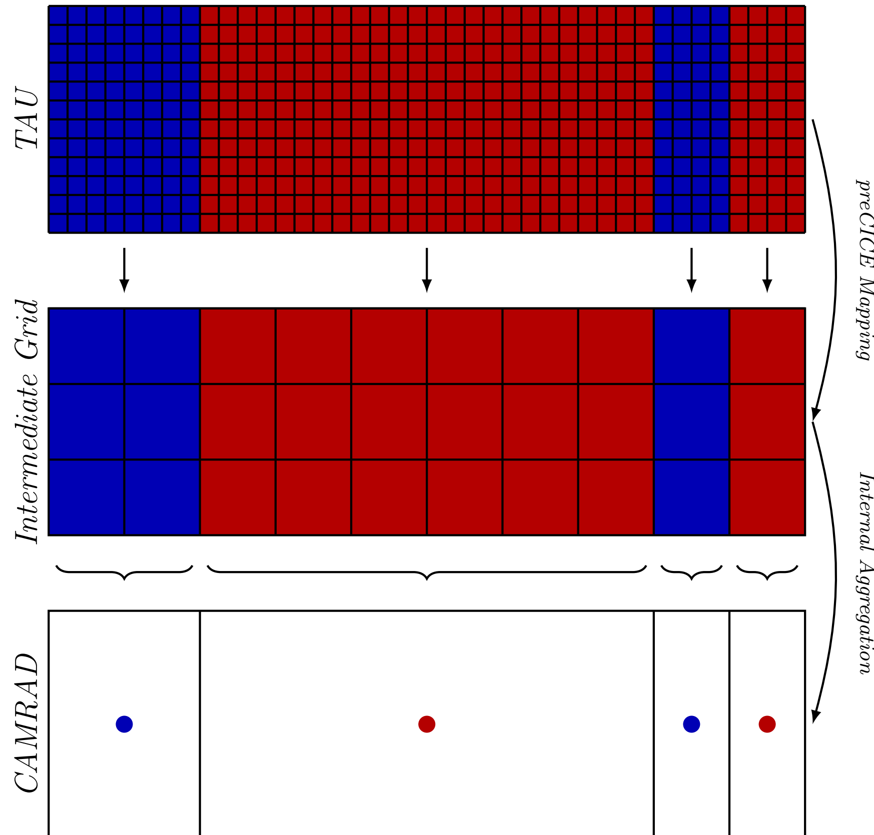


Figure 6.11: Successful Mapping of Aerodynamic Loading Data from TAU to CAMRAD II with Intermediate Grid.

The implementation of the preCICE mapping methods was tested by comparing force data aggregated in the TAU code as well as the calculated force values at the collocation points in CAMRAD II. In order to examine the accuracy of the mapped sections, we compared the summation of forces on panels using the nearest-neighbour interpolation and the RBF interpolation. The results can be seen in Fig. 6.12. We can see that the nearest-neighbor approach has far lower error for this application. This is expected: we require the force data to be cleanly divided between panels as defined in CAMRAD II. However, when using the RBF interpolation, the influence of points disregards panel edge discretisation. This leads to force values “leaking” between panels, which explains the errors when using RBF interpolation for mapping between the two solvers. Certainly, reducing the cut-off radius of the RBF mapping does increase the accuracy. However, this requires the cell size in the intermediate grid to approach the cell size

in the TAU discretization before the effect of "leaking" is resolved. Additionally, the RBF interpolation is more computationally expensive, requiring more than 10x longer to complete for the same discretisation. Hence, for this work, only nearest-neighbour interpolation was used.

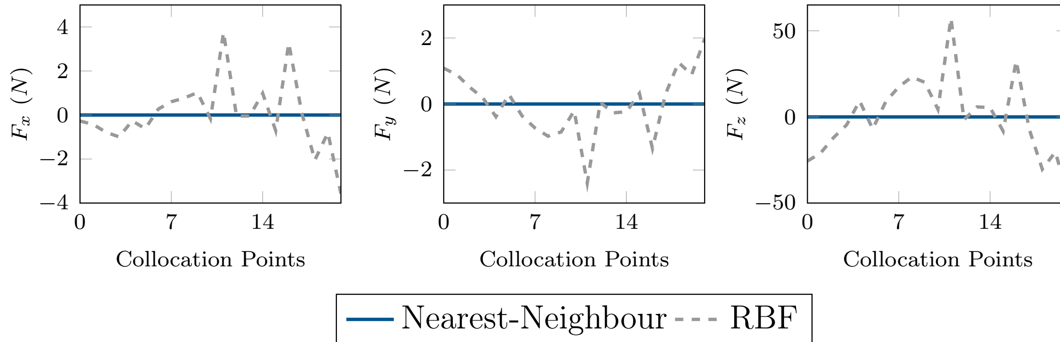
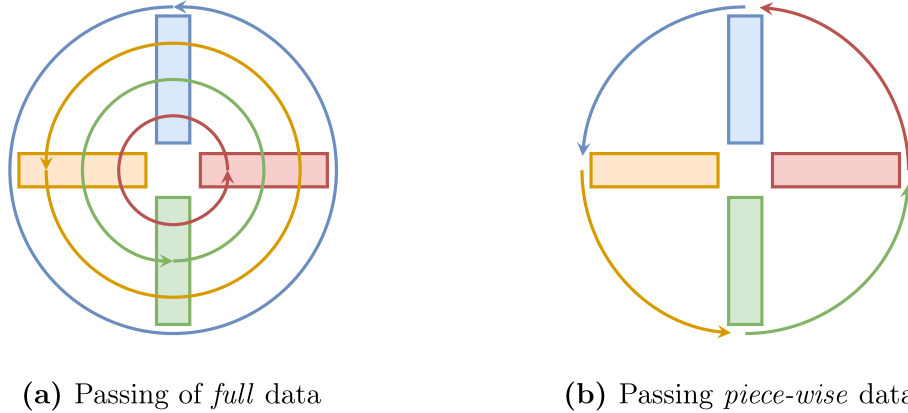


Figure 6.12: Comparison of Nearest-Neighbor Interpolation and RBF Interpolation.

The second step is to allow for the passing of a full revolution of data at each coupling step. We could try to pass a fourth dimension of data (time) via preCICE. However, this is not currently feasible as this particular functionality is not supported by preCICE; the library only allows passing data in a tightly-coupled manner. This is because we are unable to interpolate in time in the same way that field variables are interpolated between different discretizations. Instead, we treat data at each azimuth as a separate data type in the preCICE configuration. In the base simulation, we have 4 blades and 24 azimuth positions per revolution. If we approached this naively, we would then define 4 meshes (one for each blade) with 24 data types (one for each azimuth position) in the preCICE configuration, effectively passing 96 arrays of data from TAU to CAMRAD II at each coupling step.

However, we understand that the four blades in the simulation are identical, the simulation output is periodic in nature, and the fluid simulation should reach steady state before data is passed. Hence, we can, instead of passing the *full* data from 24 azimuth positions for each blade, pass data in a *piece-wise* manner.

The difference in the two methods is shown in Fig. 6.13a and 6.13b. In Fig. 6.13b, each of the 4 blades only passes data from $\frac{1}{4}$ of a revolution. This is then combined to form a full revolution of data. In order to determine if the piece-wise implementation is feasible, we implemented both methods and compared the passed force data. We ran the individual solvers to steady state solutions and passed data via preCICE using the two methods. The results are plotted in Fig. 6.14; there is no significant difference between the two methods.



(a) Passing of *full* data (b) Passing *piece-wise* data.

Figure 6.13: (a) Passing of Full data. Each blade passes data from full revolution (b) Passing of Piece-wise data. Each blade only passes data from part of a full revolution.

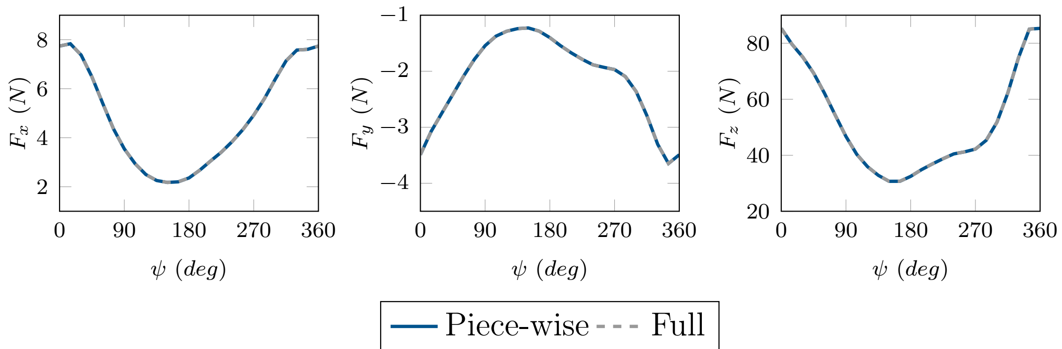


Figure 6.14: Passing of Full and Piece-wise Force Data in x- (left), y- (middle) and z-axis (right).

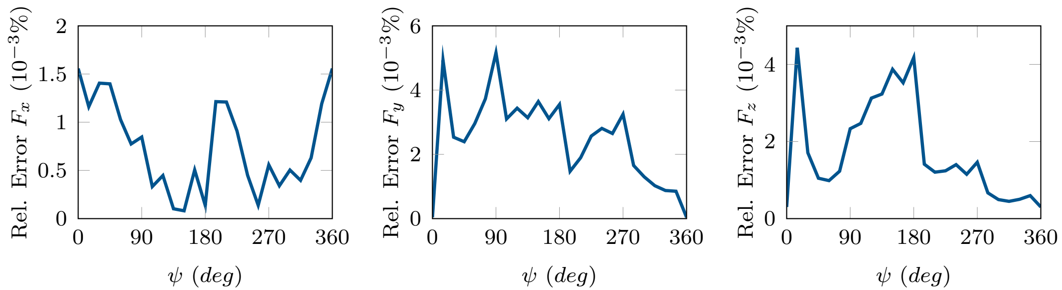


Figure 6.15: Relative Error of Full and Piece-wise Force Data in x- (left), y- (middle) and z-axis (right). Rel. Error = $(F_{\text{piece-wise}} - F_{\text{full}}) / F_{\text{full}}$.

The relative error for the forces in the x, y and z axes are shown in Fig. 6.15. The error is in the order of 10^{-3}%, which is negligible in this scenario. Hence, for the remainder of the work, employ the piece-wise methodology when passing data. This saves time and there is almost no change in the final results.

Writing the coupling configuration and preCICE configuration is a daunting

task with so many datatypes. To give an idea of the complexity of the piece-wise case, the preCICE configuration file is visualized with the preCICE config-visualizer tool in Fig. 6.16, which only shows data from a single blade fully. Each blade requires two sets of data (force and moment) for each azimuth position. This sums to a total of 50 different sets of data that must be properly defined in the preCICE, TAU and CAMRAD II configuration files for a single revolution. Ensuring the correctness of these definitions is difficult and prone to human error. Thus, configuration file generators are included in the tutorial respository containing this simulation case to automate this step.

6.2.6 Folder Structure & Running Tutorial Case

The folder structure for this simulation is almost identical to the folder structure discussed in Chapter 5. Hence, it will not be discussed in this work. Please refer to the Gitlab repository to see instructions to run this tutorial simulation.

Figure 6.16: Visualization of preCICE piece-wise configuration file. Only Data from Mesh0 is fully visualized [52].

6.3 Case 2: Loose Coupling with Deformation

After completing the simple coupling case without deformation, the next step is to implement deformation. One of the limitations of the previous coupling implementation was that it only considered rigid body motion of the blades in the fluid simulation. We attempt to remedy this by implementing a simple deformation case in the TAU solver using the passed motion information.

6.3.1 Simulation Setup

CAMRAD II Simulation Setup

The simulation setup for TAU and CAMRAD II is identical to the setup in Case 1. The CAMRAD II `Solver` is identical, as is the CAMRAD II input files, see Section 6.2.3 for more details.

TAU Simulation Setup

The TAU `HelicopterSolver` is modified to add a deformation step to the simulation loop. The updated algorithm can be seen in Algorithm 9. There are two main updates to the algorithm. First, before the running the outer loop iterations, a series of Fourier functions are created from received motion data that calculate the angular displacements at a given azimuth position. This step is seen in Line 10. Next, a new scatfile is created for each outer loop timestep and the primary grid is deformed. This occurs in Line 17 and 18.

While the steps to include deformation into the `HelicopterSolver` may seem trivial, the calculation of the displacements and the creation of the scatfiles is quite involved. The exact deformation calculations are presented in the next section.

6.3.2 Deformation Calculation

The CAMRAD II solver outputs the absolute positions of the collocation points. However, we cannot directly deform the TAU grid based on the absolute position of these collocation points; we would opt to reduce the amount of necessary deformation as much as possible to prevent badly-conditioned cells.

Instead, TAU allows for a combination of rigid body motion and grid deformation, such that grid deformation can be minimized. The approach taken in this work is to re-use the rigid body motion implemented in Section 6.2.1 and supplement the rigid body motion with grid deformation to recreate the elastic motion output by CAMRAD II.

Algorithm 9: `TAU-Python` algorithm for loosely-coupled case with deformation

```
1  init TAU-Python classes: PyPara, PySolv, PyPrep, PyDeform with TAU
   Parafile
2  init solver via PySolv
3  for t = 1, 2, ..., t_init do          ▷ t_init = number of initial iterations
4      init outer loop
5      advance time and motion in TAU simulation
6      solve multigrid
7      run inner loop n_in times  ▷ n_in = number of inner time steps in Parafile
8  receive motion and sensor position data from solid solver
9  update motion input file
10 create Fourier series for deformation
11 while self.precice_interface.is_coupling_ongoing() do
12     for t = 1, 2, ..., t_rev do  ▷ t_rev = number of iterations per coupling step
13         init outer loop
14         advance time and motion in TAU simulation
15         solve multigrid
16         run inner loop n_in times
17         create scatfile for deformation
18         deform primary grid
19         print monitoring data
20         save output data per Parafile settings
21     process and send aerodynamic loading and azimuth data to solid simu-
   lation
22     update time in preCICE coupling
23     receive and process updated motion data from solid solver
24     update motion input file
25 stop and finalize solver
26 stop and finalize preCICE coupling interface
```

Deformation in TAU is performed by the “deformation” executable or the `PyDeform` `TAU-Python` class. Both operations require the writing of a scatfile, described in Section 4.1.1. Recalling that scatfiles apply an RBF interpolation, we do not need to provide the deformation magnitudes at every node in the primary grid. Instead, the deformation methods in TAU accepts a coarser grid and interpolates values at these points. This is referred to as the *deformation grid.* The deformation grid discretisation was set to 200 nodes along the blade length and 4 nodes along the blade width, resulting in a mesh approximately 30 times coarser than the primary TAUmesh. While this discretisation produced good results (detailed in Section 7.4), a further mesh refinement study should be conducted optimize the used discretization.

The next difficulty is then to calculate the deformation magnitudes after ommitting the effects of rigid body motion on the deformation grid. In order to do so, we now have to calculate the movement of the nodes due to the rigid body motion implemented in Section 6.2.5. Recalling the hierarchy discussed in Section 4.1.3 and shown in Fig. 6.17, we have to determine the correct order to apply the rotational matrices.

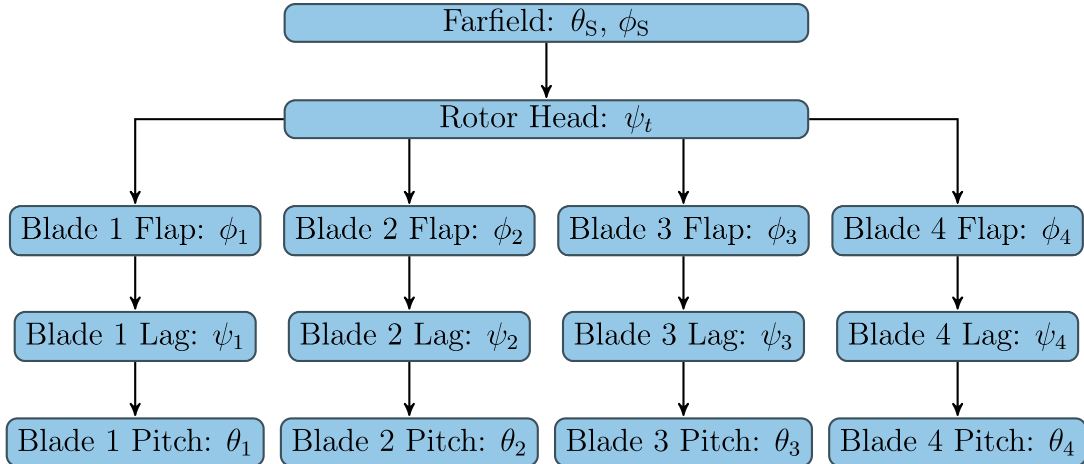


Figure 6.17: Motion Hierarchy of the 4-bladed CFD rotor blade simulation [1].

Now that we are calculating these angles instead of relying on TAU internal calculations, we have to consider timestep or azimuth position when using the Fourier series. Hence, we refer to the blade flap, lag, and pitch angles at time t_1 or azimuth position 1 as $\theta^{(t_1)}$, $\psi^{(t_1)}$, and $\phi^{(t_1)}$. The other angles listed in the motion heirarchy are constant and do not change with time in this work.

For the case of Blade 1 at $t = t_1$, we first apply the blade motion in the given order: $\theta_1^{(t_1)}$, $\psi_1^{(t_1)}$, and $\phi_1^{(t_1)}$. We then apply the helicopter blade rotation, ψ_t. We then apply the rotor shaft and pitch angles θ_S, ϕ_S. The application of the respective rotation matrices in this order then maps mesh nodes in TAU-Code-Grid frame to the Geodesic or inertial frame. Rotation matrices are defined as rotation around axes in the TAU-Code-Grid frame. We use the following notation to describe rotation matrices: the rotation matrix for an angle θ around the X-axis in the TAU-Code-Grid frame is $R_{X\theta}$, which is defined in Eq.6.2.

$$R_{X\theta} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & cos(\theta) & sin(\theta) \\ 0 & -sin(\theta) & cos(\theta) \end{bmatrix} \tag{6.2}$$

Thus, we can represent the application of these matrices mapping coordinates on Blade 1 in the TAU-Code-Grid frame to the Geodesic frame in Eq. 6.3. We

simplify this long string of rotation matrices into a single *motion matrix* $M^{(t_1)}$.

$$x_G = \underbrace{R_{X\phi_S} R_{Y\theta_S} R_{Z\psi_t} R_{X\phi_1^{(t_1)}} R_{Z\psi_1^{(t_1)}} R_{Y\theta_1^{(t_1)}}}_{M^{(t_1)}} x_\tau \tag{6.3}$$

Now that we understand how rigid body motion is implemented, we can then similarly convert the deformation into a series of such matrices. The method is identical. However, unlike the rigid body motion, where the Fourier coefficients from the tip of the rotor blade are applied to the entire rotor blade, we now generate a different set of blade motion angles at each point along the rotor blade. Thus, as seen in Eq. 6.4, we can calculate a specific *deformation matrix* D_i for an arbitrary TAU node x_i with the specific blade motion angles $(\theta_{(i)})_1$, $(\psi_{(i)})_1$ and $(\phi_{(i)})_1$.

$$(x_{(i)})_{\text{deformed}} = \underbrace{R_{X\phi_S} R_{Y\theta_S} R_{Z\psi_t} R_{X(\phi_{(i)})_1^{(t_1)}} R_{Z(\psi_{(i)})_1^{(t_1)}} R_{Y(\theta_{(i)})_1^{(t_1)}}}_{D_i^{(t_1)}} (x_{(i)})_\tau \tag{6.4}$$

However, the CAMRAD II output does not have the same discretisation as the deformation grid. Hence, we cannot immediately determine the specific blade motion angles required for each point. Instead, the flap, lag and pitch angles are calculated at each CAMRAD II collocation point and linearly interpolated for each point in the deformation grid.

Putting everything together, we can then calculate the displacement $\delta^{(t_1)}$ at $t = t_1$ for one node $x_{(i)}$ as follows:

$$\begin{aligned} M\delta^{(t_1)} &= D_i^{(t_1)} x_i - M^{(t_1)} x_i, \\ \delta^{(t_1)} &= (M^{(t_1)})^{-1} D^{(t_1)} x - x, \\ &= \underbrace{\left((M^{(t_1)})^{-1} D_i^{(t_1)} - I \right)}_{Q} x_i. \end{aligned} \tag{6.5}$$

We call the final matrix required to calculate $\delta^{(t_1)}$ directly Q. As Q is a 3×3 matrix, when we assemble the deformation matrices for each point in the deformation grid, we get a sparse block-wise diagonal matrix, which we can exploit to perform this calculation step efficiently.

6.3.3 Folder Structure & Running Tutorial Case

The TAU folder structure for this simulation is slightly different from the simulation folder structures introduced in Chapter 6. While the CAMRAD II folder structure remains the same. For the TAU folder structure, an additional `tau_precice` folder is used to store newly introduced classes and subclassed `HelicopterSolver` and `HelicopterConfig` classes. The new TAU folder structure is shown in Fig. 6.18.

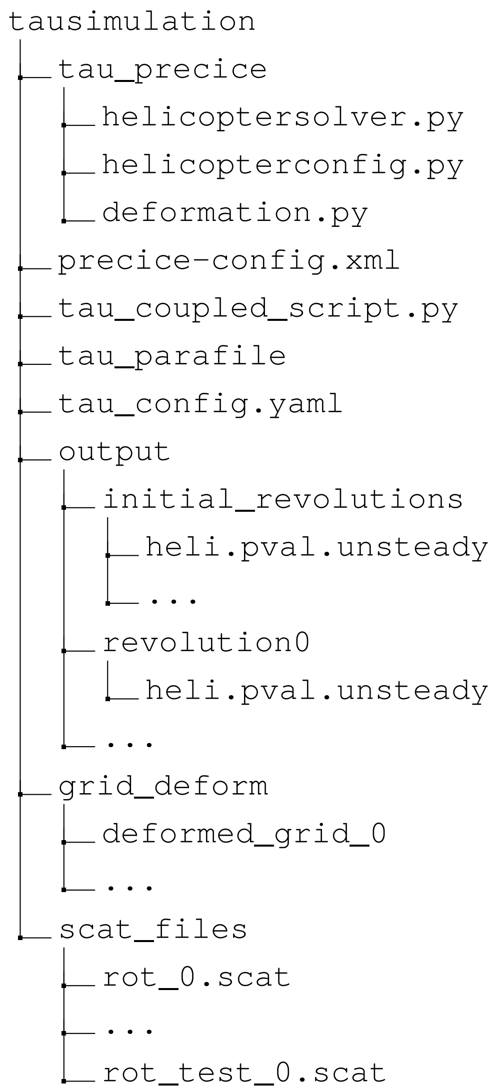


Figure 6.18: Isolated Rotor Blade Simulation with Deformation Folder Structure.

A new `deformation.py` script is introduced. This contains a custom `Deformation` class which calculates the deformation described in Eq. 6.5 and writes out a scatfile in the `scat_files` folder. The reason that deformation cannot be handled generically is because deformation settings are stored in blocks in the TAU parafiles. Ensuring that the correct deformation is calculated, the correct input block in the parafile is updated and that the deformation occurs at the correct position in the simulation loop is highly variable. Hence, for this case, a new `HelicopterSolver` class was created and introduced as new `Solver` subclass.

7. Results

This section discusses the results from the implementations discussed last chapter. The exact code used to produce these results is listed in Appendix D.

7.1 Toy Example: Perpendicular Flap

The perpendicular flap was run with the input parameters listed in Section 6.1. A snapshot at time $t = 2.0s$ showing the pressure distribution can be seen in Fig. 7.1. The displacement of a watchpoint located at the tip of the perpendicular flap is tracked in the original `OpenFOAM-CalculiX` coupled simulation. The same watchpoint was tracked in the TAU-`CalculiX` simulation and the results are plotted in Figs. 7.2.

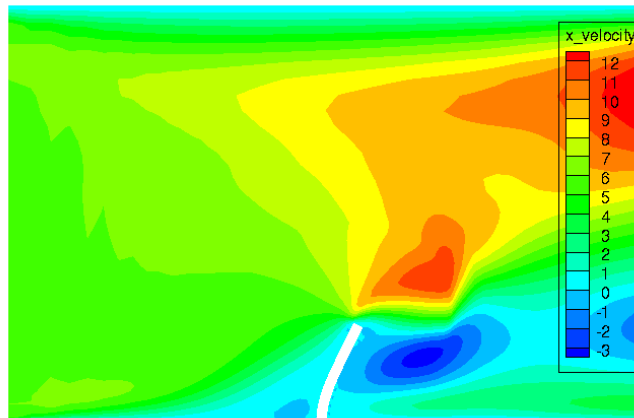


Figure 7.1: Snapshot of TAU-CalculiX Toy Example.

We can see that the TAU-`CalculiX` simulation has not been tuned to match the `OpenFOAM-CalculiX` simulation. Hence, the flow conditions of the two simulations are different. This can be seen from the different amplitudes of the displacements in the new adapter and the original coupling. However, we can see that the overall movement of the two flaps are similar: the flap bends in the direction of the flow before returning to a similar original position. Thus, we can see that the passing of data works in this coupling.

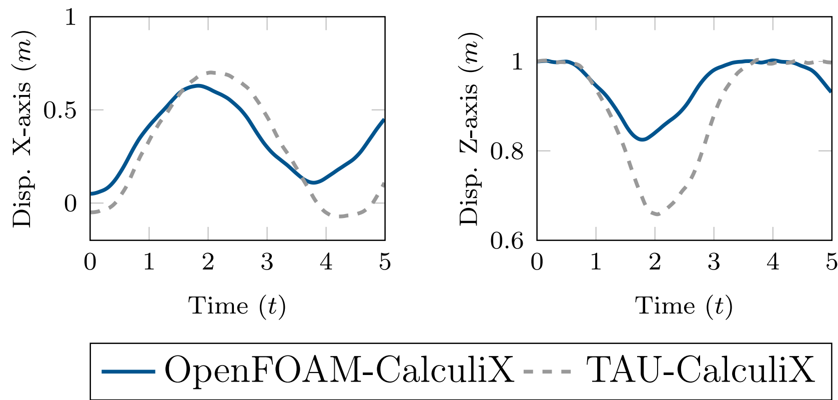


Figure 7.2: Displacements of watchpoint tracked in TAU-`CalculiX` and `OpenFOAM-CalculiX` in X-axis (left) and Z-axis (right).

7.2 Case 1: Loose Coupling without Deformation

The results of the loosely-coupled simulation without deformation are described in this section. As mentioned in Section 6.2, a total of seven initial revolutions were run prior to coupling. This was to remove unphysical vortices around the rotor blades and increase the stability of the coupling. The flowfield around the rotor blades is visualized in the Fig. 7.3.

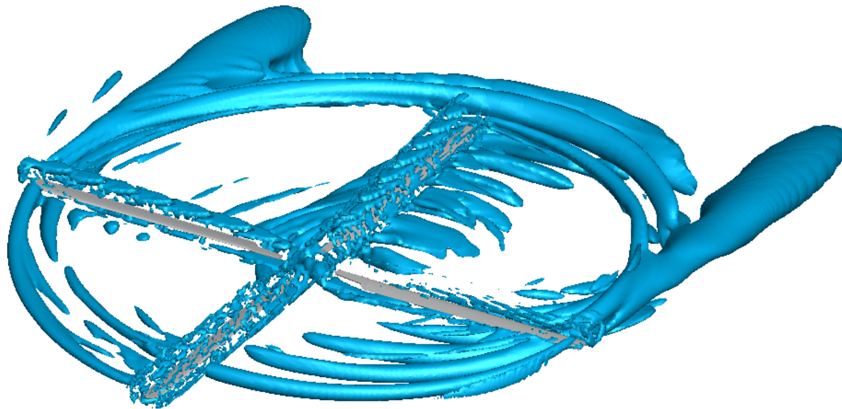


Figure 7.3: Flowfield Around Rotor Blades.

The coupled simulation ran for a total for four coupling iterations. Three sets of data are shown in this section, the data prior to coupling, the data after four coupling iterations, and a table of the CAMRAD II control values. Firstly, the thrust distribution prior to coupling across a full revolution are shown in Fig. 7.4. The force and moments values at the collocation point located at $r = 0.87$ for both TAU and CAMRAD II simulations are presented in Fig. 7.5.

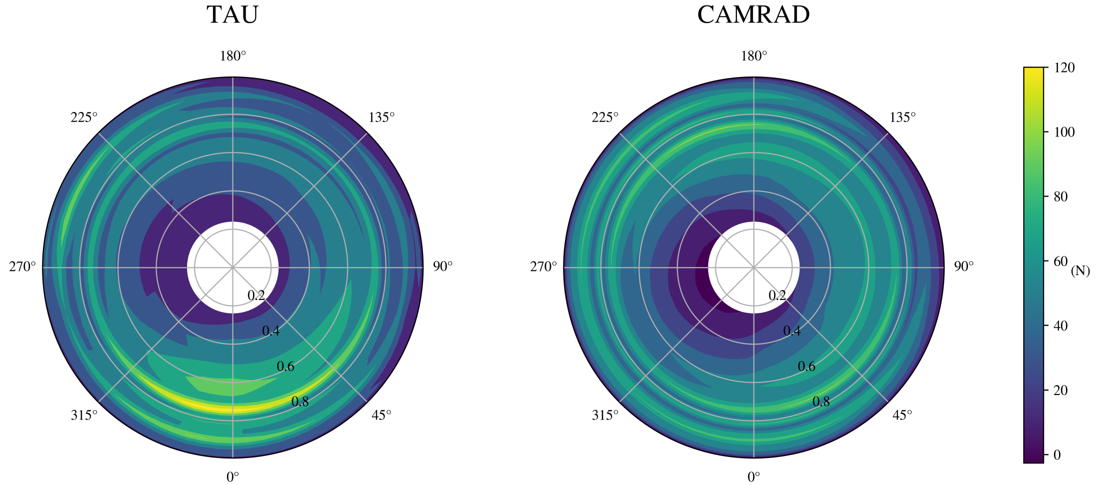


Figure 7.4: Comparison of TAU (left) and CAMRAD II (right) thrust distribution (F_z) over a single revolution prior to coupling.

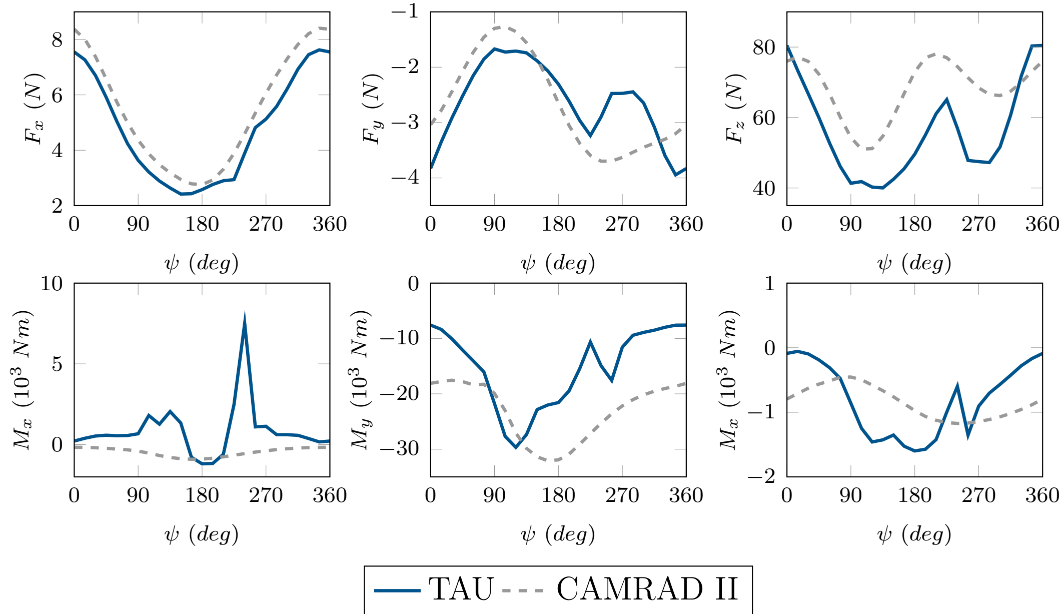


Figure 7.5: Comparison of TAU and CAMRAD II aerodynamic force and moment data over a single revolution prior to coupling at CAMRAD II sensor position $r = 0.87$. Force data in x-, y-, z-axis from left to right of top row. Moment data in the x-, y-, z- axis from left to right of bottom row.

After four coupling iterations, the simulation tends towards convergence. We can see that the general thrust distribution of the TAU and CAMRAD II simulation are quite similar in Fig. 7.6. Additionally, the force and moment data are in good agreement in Fig. 7.7. However, we can observe some oscillating behavior of subsequent coupling steps. This can be observed when examining the changes in the control inputs shown in Table 7.1. We see that after the initial

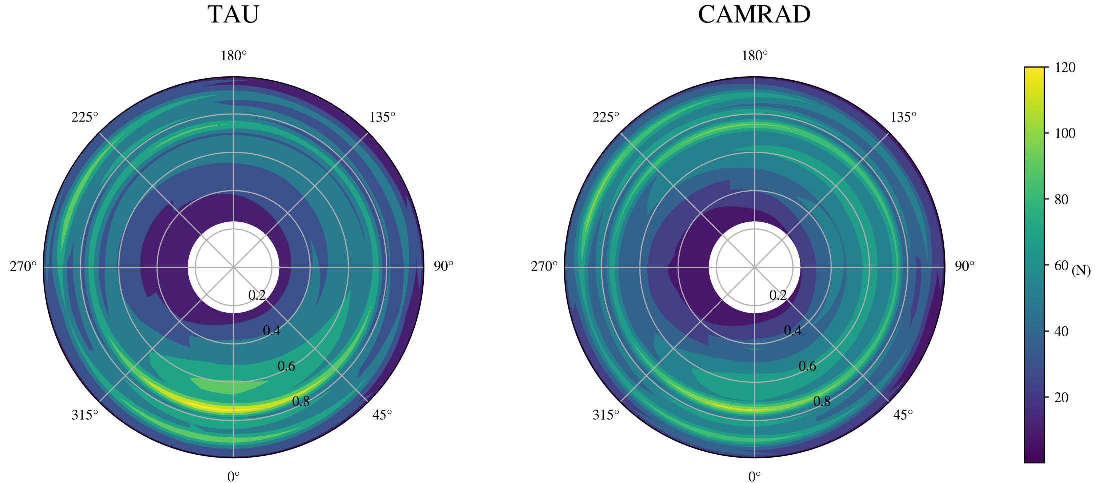


Figure 7.6: Comparison of TAU (left) and CAMRAD II (right) thrust distribution (F_z) over a single revolution after 4 coupling iterations.

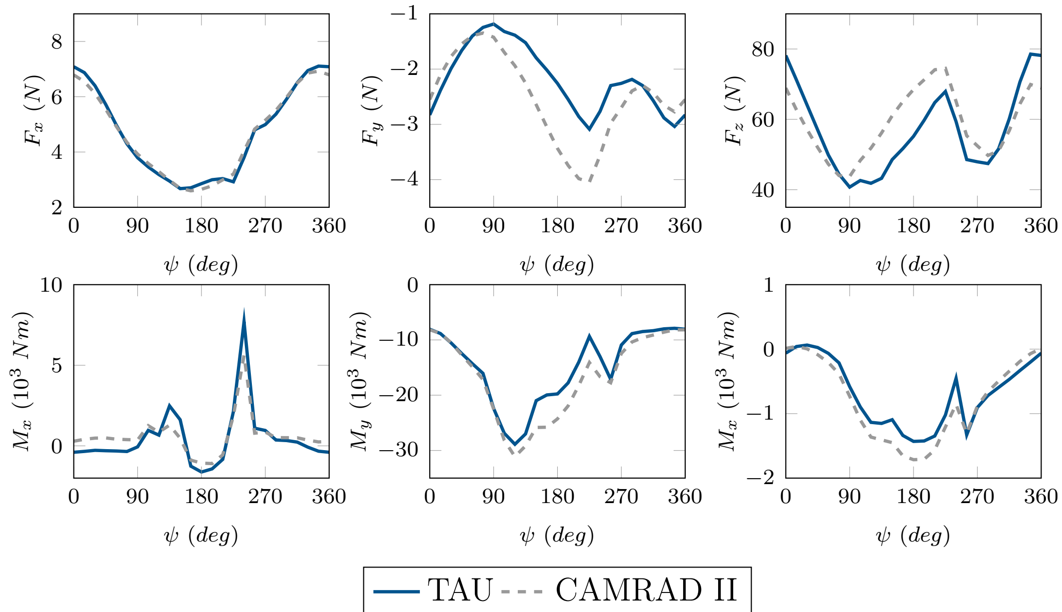


Figure 7.7: Comparison of TAU and CAMRAD II aerodynamic force and moment data over a single revolution after 4 coupling iteration at CAMRAD II sensor position $r = 0.87$. Force data in x-, y-, z-axis from left to right of top row. Moment data in the x-, y-, z- axis from left to right of bottom row.

shift from the first coupling step, subsequent steps oscillating with a decreasing amplitude towards convergence.

This oscillating behavior is reminiscent of explicit time-stepping schemes with overly large timesteps, which leads to the solver overcorrecting when a new gradient is calculated instead of quickly reaching convergence. While there was insufficient time to fully experiment with input parameters to mitigate this

Table 7.1: Overview of CAMRAD II control values per coupling iteration.

Iteration	$\Delta\Theta_{\text{coll}}$	$\Delta\Theta_{\text{lat}}$	$\Delta\Theta_{\text{long}}$	Θ_{coll}	Θ_{lat}	Θ_{long}
N/A	N/A	N/A	N/A	6.45	−2.03	2.69
1	−0.1	0.9	−0.5	6.35	−1.13	2.19
2	0.1	−0.6	0.37	6.45	−1.73	2.56
3	−0.06	0.41	−0.18	6.39	−1.32	2.34
4	0.03	−0.27	0.15	6.42	−1.59	2.49
Units	deg	deg	deg	deg	deg	deg

phenomenon, several promising avenues can be explored. Firstly, we could reduce the relaxation factor in the coupling code when passing aerodynamic loads from TAU to CAMRAD II and when passing motion data from CAMRAD II to TAU. This would reduce the effect of the passed forces and motions which could prevent the oscillations seen. However, these specific relaxation factors would require further experimentation.

Additionally, it is worth exploring implicit coupling schemes in the preCICE coupling as some convergence of boundary conditions could improve the stability of the coupled simulation. However, this would required further implementation of checkpointing in the TAU adapter, which is currently not supported. Moreover, implicit coupling schemes may increase the number of time the TAU solver would need to run, which could increase the time to solution.

7.3 Profiling

After examining the output of the adapter, the TAU adapter was profiled against the original base code. Both codes were run on the cluster in the Chair of Helicopter Technologies on node TUM-WE140. Both codes were run with identical preprocessing with 24 processes. Both codes were run with zero inner iterations as the simulation loops were largely identical. The two codes were allowed run for two full coupling iterations.

The profiles of the orginal script, the TAU adapter and the CAMRAD II adapter generated are seen in Tables 7.2, 7.3 and 7.4 respectively. The output has been sorted based on cumulative time to allow an easy comparison of the main methods called to initialize and couple the simulations.

We see that the runtime of the solvers is roughly the same. The original code runs for approximately 300 minutes ($1.772\dot{1}0^4$ seconds) and the TAU adapter running for approximately 257 minutes ($1.541\dot{1}0^4$ seconds). The solver loop

Table 7.2: Profile of original script.

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
1	0.0796	0.0796	1.84e+04	1.84e+04	coupled.py:31(<module>)
2	1.772e+04	8862	1.772e+04	8862	:0(<posix.waitpid>)
2	0.000194	9.7e-05	1.772e+04	8862	subprocess.py:1379(wait)
2	0.000185	9.25e-05	1.772e+04	8862	subprocess.py:514(call)
4	9e-05	2.25e-05	1.772e+04	4431	subprocess.py:473(_eintr_retry_call)
2	4.5e-05	2.25e-05	1.772e+04	8862	subprocess.py:525(check_call)
2	0.04564	0.02282	653.2	326.6	tau_functions.py:398(tau_rotor_forces_moments)
96	228.4	2.379	645	6.718	tau_functions.py:253(tau_blade_forces_moments)
2739648	146.4	5.343e-05	396	0.0001446	numeric.py:1701(cross)
8218944	61.96	7.539e-06	211.5	2.574e-05	numeric.py:1620(moveaxis)

Table 7.3: Profile of TAU adapter script.

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
1	0.3075	0.3075	1.541e+04	1.541e+04	tau_coupled_script.py:18(<module>)
1	6.2e-05	6.2e-05	1.541e+04	1.541e+04	tau_coupled_script.py:28(main)
1	0.001427	0.001427	1.519e+04	1.519e+04	helicoptersolver.py:403(execute_precice)
1	3.5e-05	3.5e-05	1.519e+04	1.519e+04	adapter.py:160(execute)
3	0.101	0.03366	1.516e+04	5055	helicoptersolver.py:516(tau_simulation_loop)
720	1.5e+04	20.84	1.5e+04	20.84	:0(<_tau_python.tau_solver_unsteady_advance_motion>)
1	9.6e-05	9.6e-05	216.9	216.9	adapter.py:69(initialize)
4	0.001176	0.000294	178.4	44.59	PySolv.py:88(init)
1	0.001011	0.001011	154.8	154.8	helicoptersolver.py:60(__init__)
1	0.000114	0.000114	154.8	154.8	adapter.py:85(initialize_solver)
4	90.31	22.58	90.31	22.58	:0(<_tau_python.tau_solver_init_params>)
720	72.51	0.1007	72.51	0.1007	:0(<_tau_python.tau_solver_write_output_conditional>)
1	0.2093	0.2093	61.79	61.79	helicoptersolver.py:131(setup_data_handlers)
4	51.05	12.76	51.1	12.77	helicoptersolver.py:345(filter_surface_nodes)
...	...	...	...	...	...
2	0.000693	0.0003465	19.33	9.667	interface.py:107(write_all)
...	...	...	...	...	...
1	2.826	2.826	2.826	2.826	:0(<method 'initialize' of 'precice.Interface' objects>)
...	...	...	...	...	...
2	7.9e-05	3.95e-05	0.02105	0.01053	interface.py:76(read_all)

Table 7.4: Profile of CAMRAD II adapter script.

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
1	0.8984	0.8984	1.541e+04	1.541e+04	camrad_coupled_script.py:18(<module>)
2	1.519e+04	7594	1.519e+04	7594	:0(<method 'advance' of 'precice.Interface' objects>)
1	218.5	218.5	218.5	218.5	:0(<method 'initialize' of 'precice.Interface' objects>)
1	0.05565	0.05565	1.585	1.585	__init__.py:106(<module>)
4	0.000671	0.0001677	1.308	0.3271	camrad_functions.py:182(camrad_blade_forces_moments)
1	0.06242	0.06242	1.304	1.304	camrad_utility.py:11(<module>)
24	0.7455	0.03106	1.052	0.04383	camrad_functions.py:83(read_sensor_data)
1	0.2736	0.2736	0.8952	0.8952	__init__.py:1(<module>)
1	0.8764	0.8764	0.8764	0.8764	:0(<method 'finalize' of 'precice.Interface' objects>)
1	0.0258	0.0258	0.8112	0.8112	__init__.py:93(<module>)

in the TAU adapter may be slightly optimized by using newer `TAU-Python` methods, which may account for the difference in solver runtime. However, the original code runs the solver in a subprocess and we cannot clearly analyze the reasons for the differences in the solver time. This difference should also be ignored as it does not involve the adapter code and native TAU users should be able to optimize their simulation loops.

Instead, we look at the changes introduced by the new adapter code. This is separated into two categories: setup time and coupling time. The former refers to the time at startup required to setup a simulation and the latter refers to the time required to process and pass data between coupled solvers.

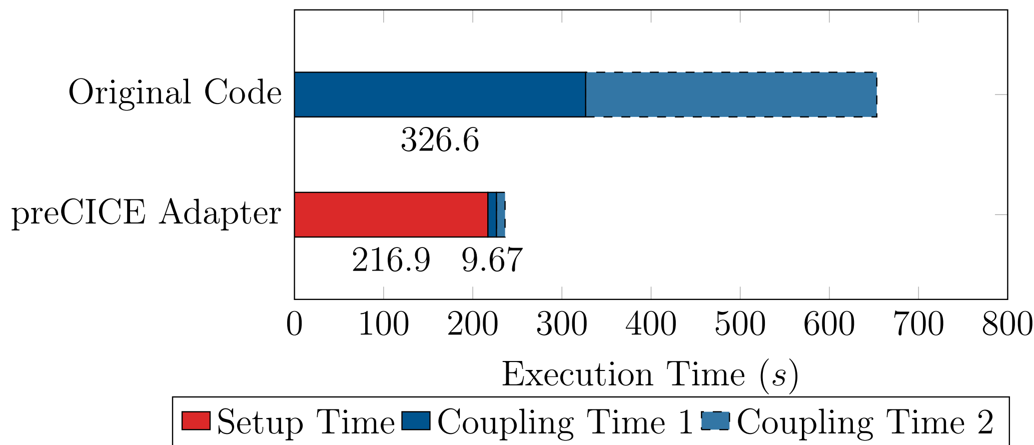


Figure 7.8: Bar graph showing setup time and coupling time.

Of the two preCICE adapters, the TAU adapter requires a longer setup time and requires more computation during each coupling step; the CAMRAD II adapter always waits for the TAU adapter when setting up or coupling. Hence, as the TAU adapter is the slower adapter, we can compare it against the original coupling to determine how performance has changed in the new code. A graph of the cumulative setup and coupling time of the TAU adapter and the original coupling is shown in Fig. 7.8.

The first difference is that the original code shows no initialization time. This is because both CAMRAD II and TAU are run inside subprocesses and this information is difficult to extract. On the other hand, we can determine the setup time of the adapter code. The TAU adapter requires about 216 seconds to run the `initialize()` method of the `Adapter` class. This time, indicated in red in Fig. 7.8, includes partitioning the TAU grid and setting up the preCICE meshes (approximately 61 seconds); instantiating `TAU-Python` classes (approximately 154 seconds); and initializing preCICE (approximately 2.8 seconds).

Despite this unfair comparison (as the setup time in the original script is effectively hidden), we see that the adapter code actually runs faster within one coupling iteration. The main method called in the original coupled script to calculate aerodynamic loads is `tau_rotor_forces_moments()`. The method takes approximately 300 seconds per call (shown in blue in Fig. 7.8). The original code performs a gather step to combine the outputs from multiple

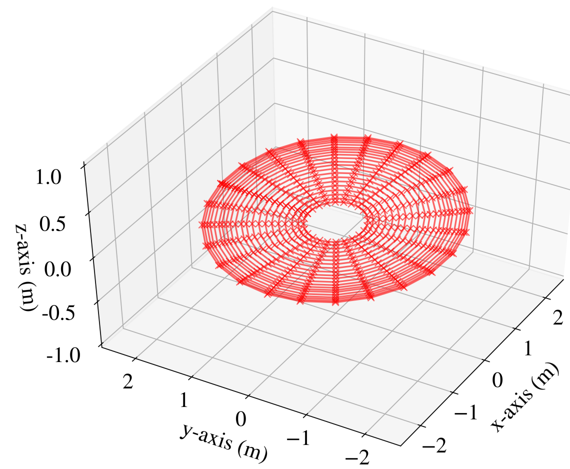
parallel processes before processing data in a sequential and non-vectorized manner. The TAU adapter skips this step and instead processes data in parallel. Additionally, the mathematical method used in the original code was vectorized and simplified in the TAU adapter. As a result, the reading and writing of all passed data (including the calculation and passing of non mesh-based data) takes less than 10 seconds per coupling iteration in the new adapter code.

This speedup does appear impressive at first glance but a coupled rotor blade simulation can run for many hours, even days. Thus, when we put things into perspective, the actual performance gain by using the adapter code is not significant. However, we can clearly show that introducing preCICE does not impede performance of the coupled simulation.

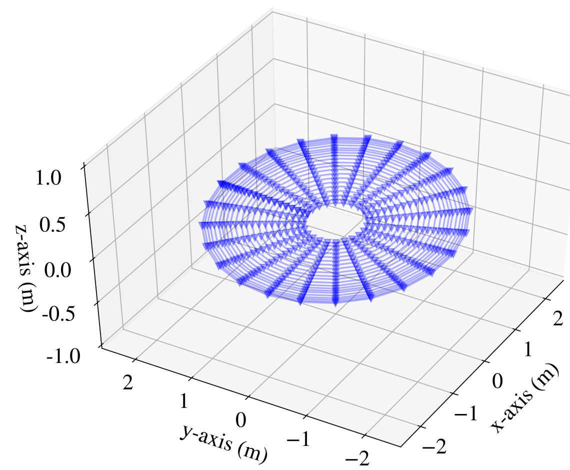
7.4 Case 2: Loose Coupling with Deformation

A study was conducted to ensure that the applied deformation was an accurate reflection of the CAMRAD II output. There are two points that required verification. Firstly, we need to determine if the angular approximation of the displacement is accurate. Secondly, we have to examine if the applied deformation in the TAU output files occurs. A coupled TAU simulation was run on the LRZ to produce deformed grids that are compared with the deformation output from CAMRAD II. The coupled simulation was run for one coupling step and the examine data compares the output of the *first* coupled iteration.

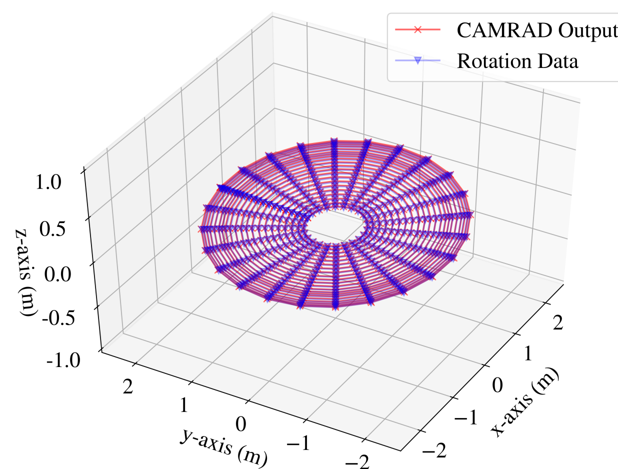
The first study creates a 1D discretisation matching the sensor output positions in CAMRAD II and applies the displacements prescribed in the CAMRAD II output files. This is the CAMRAD II collocation points in the blade local frame per the CAMRAD II simulation. Next, we apply the rotation matrices calculated per Eq. 6.5 and compare the two results. We see the output from CAMRAD II and the effect of the rotation matrices in Fig. 7.9, the CAMRAD II ouput is seen in Fig. 7.9a, the positions calculated via rotation matrices is seen in Fig. 7.9b. The two plots are overlayed in Fig. 7.9c.



(a) CAMRAD II collocation point position over one revolution .

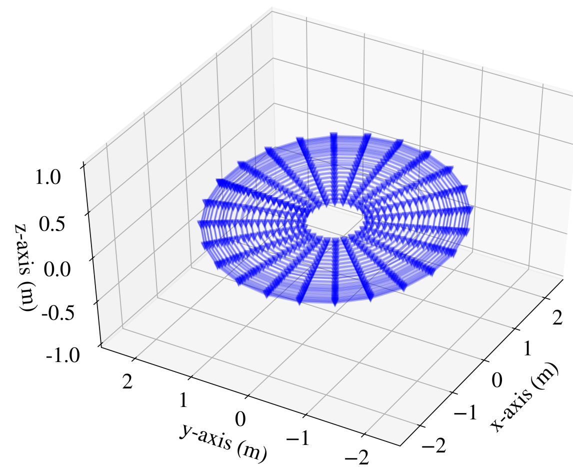


(b) Collocation point calculated with rotational matrices over one revolution.

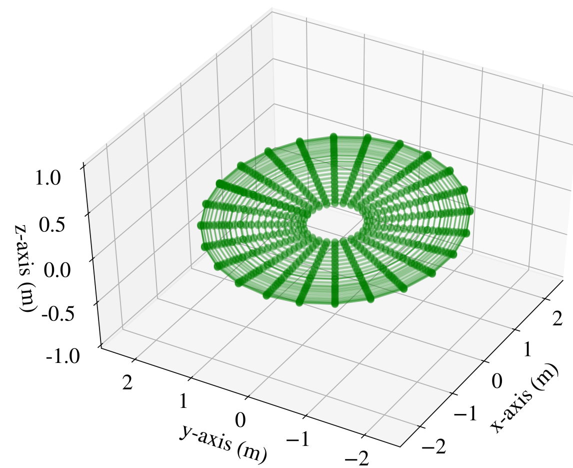


(c) Overlay of CAMRAD II collocation point position and calculated collocation point position.

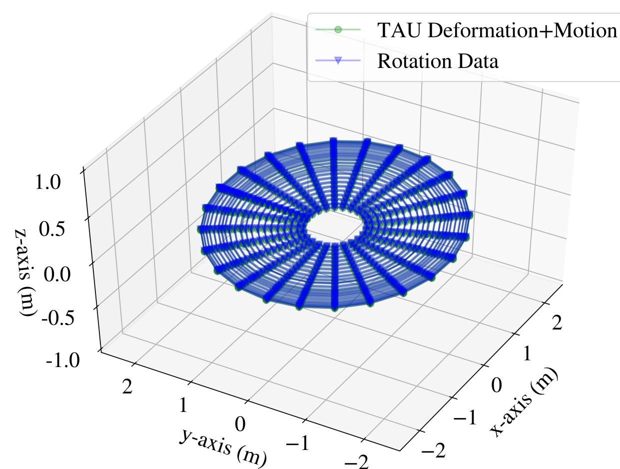
Figure 7.9: Comparison of CAMRAD II collocation point position and calculated collocation point position.



(a) Collocation point calculated with rotational matrices over one revolution.



(b) TAU deformation + motion over one revolution.



(c) Overlay of calculated collocation point position and TAU deformation + motion.

Figure 7.10: Comparison of calculated collocation point position and TAU deformation + motion.

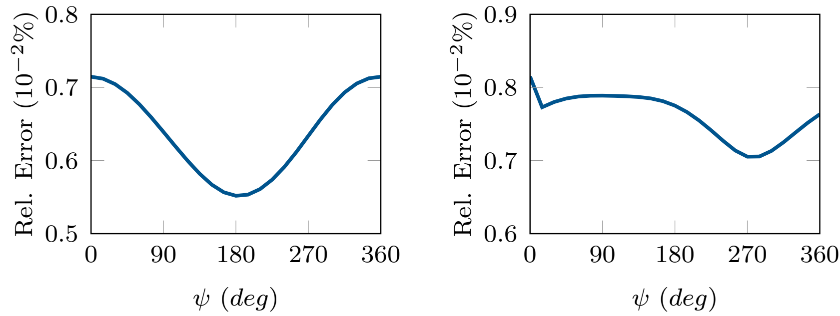


Figure 7.11: Average Relative Error per Azimuth position (left) for CAMRAD II collocation point position and calculated collocation point position. Average Relative Error per Azimuth position (right) for calculated collocation point position and TAU deformation+motion.

The relative error was calculated for each azimuth position and the average relative error for each azimuth position was plotted in Fig. 7.11. We see that the relative error for each point as in the order of 0.01%. Therefore, the use of rotational matrices is a good approximation of the deformation prescribed by CAMRAD II.

Next, we generate the scatfiles per the method described in Section 6.3.2. For this case, we gather the node ids of nodes in the TAU grid which are as close to the collocation points in the CAMRAD II descretization as possible. We take note that there is a surface at the top of the rotor blade and one at the bottom which is deformed. Hence, we gather two nodes per collocation point. We then apply the rotation matrix D to these points and compare it to the deformed TAU grid, which sees both rigid body motion and the effects of the scatfile deformation. The results are presented below in a similar fashion to the rotational matrices above.

We see that the average relative error in Fig. 7.11 at each node is in the order of 0.01%. Thus, the application of deformation via scatfile and rigid body motion provides a good approximation of the deformation prescribed by CAMRAD II.

8. Conclusion

In response to a need for a modular, extensible and easy to use solution to implement coupled Fluid-Structure Interaction (FSI) simulations, preCICE adapters for the TAU fluid solver and CAMRAD II solid solver were developed. These adapters facilitate the future implementation of more complex, multi-participant coupled simulations. This was a multi-step development process that involved working around several limitations presented by the two solvers and the employed coupling library preCICE.

Looking at the solvers, one of the main limitations with TAU and CAMRAD II is the fact that both are closed-source and do not allow modification of the source code. Hence, a different approach had to be taken to implement preCICE adapters. These are detailed in Sections 4.1 and 4.2.

For CAMRAD II, it was not possible to interact with the solver at all. Instead, a script was written that would read CAMRAD II output from output files, post- and pre-process output and incoming data and write new inputs to CAMRAD II input files. For TAU, greater customizability was possible using the `TAU-Python` API and the adapter combined calls to `TAU-Python` and preCICE. As a result, an adapter structure with an improved workflow (detailed in Section 5.3) was designed.

This structure centered around the `Adapter` class (for both TAU and CAMRAD II solvers), which functions as the user interface for the preCICE adapters. The `DataHandler` and `Solver` base classes were developed for the TAU adapter. They allow users to customize pre- or post-processing steps and design specific simulation loops in TAU (detailed in Section 5.4). Similar modularity is provided in the CAMRAD II adapter using a slightly modified `DataHandler` class (detailed in Section 5.5).

Once the basic adapter structure was developed, three cases were implemented. The first case is based off a simple tightly-coupled `OpenFOAM`-`CalculiX` simulation and the `OpenFOAM` adapter was replaced with the newly developed TAU adapter. In both cases, we were able to observe fluid-structure interaction

due to the communication of the coupled solvers. This serves as a proof of concept for the TAU adapter. However, as the flow conditions of the original and the new coupling differ, a direct comparison between the results of the two couplings is not valid.

Once this case was implemented, we moved on to the loosely-coupled simulation developed by Aaron Carnefix [1]. This implementation was more complex and posed several challenges due to the limitations of the preCICE library. These mainly included the passing of non mesh-based data, the exchange of period data in a loose coupling and the 1D-3D mapping of data. While these are known problems in preCICE, the developed solutions would not be available before the end of this work. Hence, stopgap measures were developed to address them. These challenges are discussed in greater detail in Section 6.2.5.

This coupled simulation was run for four coupling iterations and the results showed good agreement. This was shown by comparing the aerodynamic forces and moments in TAU and CAMRAD II. Additionally, this case profiled against the original code and the results showed that the introduction of the adapter code did not impede performance.

In the final implemented case, we introduced grid deformation to simulate elastic blade motion. This was accomplished by splitting the CAMRAD II output into rigid body motion and elastic motion in TAU. We then compared the motion in CAMRAD II with the deformed grids in TAU to determine the accuracy of this methodology. The relative error was observed to be in the order of 0.01%. Hence, we can conclude that this method is able to replicate the elastic motion from CAMRAD II with high accuracy in TAU.

The TAU and CAMRAD II adapters developed in this work function as a preliminary prototype and have many areas for improvement. There are two areas where work is still needed: user-friendliness of the TAU and CAMRAD II adapters and efficiency of the adapter code.

Regarding the user-friendliness of the adapters, most of the methods for producing extensible code were worked on by a single person. Hence, the code still requires users to experiment with the features and provide feedback to improve usability and functionality. This project would be greatly aided by such developments as it continues to mature.

Finally, implementation of the code in the `Datahandlers` focused on functionality as opposed to efficiency. There are a great many ways to improve the speed of calculation in the code to decrease time to solution. While most of the mathematical operations were vectorized via the Python numpy library, there

are definitely more efficient methods to perform these calculations. Similarly, work can be done to use memory buffers when accessing TAU boundary markers or output data, which would vastly improve computation speed by removing the need to read and write output files to the hard drive. While workarounds can be found to address these issues, a more direct approach would be to modify the `TAU-Python` API directly to increase useability.

References

[1] A. Carnefix, “Development of a framework for trimmed rotorcraft simulations using loose coupling of CFD and CSD software,” Master’s thesis, Technical University of Munich, 2017.

[2] D. Risseeuw, “Fluid structure interaction modelling of flapping wings,” Ph.D. dissertation, 01 2019.

[3] O. A. Bauchau, C. L. Bottasso, and Y. G. Nikishkov, “Modeling rotorcraft dynamics with finite element multibody procedures,” *Math. Comput. Model.*, vol. 33, no. 10-11, pp. 1113–1137, May 2001. [Online]. Available: http://dx.doi.org/10.1016/S0895-7177(00)00303-4

[4] L. Ahaus, S. Makinen, T. Meadowcroft, H. Tadghighi, L. Sankar, and J. Baeder, “Assessment of CFD/CSD analytical tools for improved rotor loads,” *Annual Forum Proceedings - AHS International*, vol. 2, pp. 1164–1186, 01 2015.

[5] W. Johnson, “A history of rotorcraft comprehensive analyses,” *NASA TP 2012-216012*, 04 2012.

[6] H. Saberi, M. Khoshlahjeh, R. Ormiston, and M. Rutkowski, “Overview of RCAS and application to advanced rotorcraft problems,” *AHS International 4th Decennial Specialists’ Conference on Aeromechanics*, pp. 741–781, 01 2004.

[7] O. Bauchau, “Computational schemes for flexible, nonlinear multi-body systems,” *Multibody System Dynamics - MULTIBODY SYST DYN*, vol. 2, pp. 169–225, 06 1998.

[8] C. Tung, F. Caradonna, and W. Johnson, “The prediction of transonic flow on an advancing rotor,” *Journal of the American Helicopter Society*, vol. 31, no. 2, pp. 4–9, 07 1984.

[9] K.-C. Kim and I. Chopra, "Effects of three-dimensional aerodynamics on blade response and loads," *AIAA Journal*, vol. 29, no. 7, pp. 1041–1050, 1991. [Online]. Available: https://doi.org/10.2514/3.10702

[10] C.-Y. Chow, I.-C. Chang, and L.-M. Gea, "Transonic aeroelasticity analysis for rotor blades," *Journal of Aircraft*, vol. 29, 02 1989.

[11] R. Strawn, A. Desopper, J. Miller, and A. Jones, "Correlation of puma airloads: Evaluation of CFD prediction methods," 09 1989.

[12] O. Bauchau and J. Ahmad, *Advanced CFD and CSD methods for multidisciplinary applications in rotorcraft problems.* [Online]. Available: https://arc.aiaa.org/doi/abs/10.2514/6.1996-4151

[13] G. Guruswamy, "ENSAERO - a multidisciplinary program for fluid/structural interaction studies of aerospace vehicles," *Computing Sytems in Engineering*, vol. 1, pp. 237–256, 1990.

[14] J. Alonso, L. Martinelli, and A. Jameson, "A multigrid unsteady navier-stokes calculations with aeroelastic applications," *AIAA Paper 95-0048*, 1995.

[15] W. Bousman, C. Young, F. Toulmay, N. Gilbert, R. Strawn, J. Miller, T. Maier, M. Costes, and P. Beaumier, "A comparison of lifting-line and CFD methods with flight test data from a research puma helicopter," 11 1996.

[16] K. Pahlke and B. van der Wall, "Calculation of multibladed rotors in high-speed forward flight with weak fluid-structure-coupling," 09 2001.

[17] G. Servera, P. Beaumier, and M. Costes, "A weak coupling method between the dynamics code HOST and the 3d unsteady euler code WAVES," *Aerospace Science and Technology - AEROSP SCI TECHNOL*, vol. 5, pp. 397–408, 09 2001.

[18] M. Potsdam, H. Yeo, and W. Johnson, "Rotor airloads prediction using loose aerodynamic/structural coupling," *Journal of Aircraft*, vol. 43, no. 3, pp. 732–742, 2006. [Online]. Available: https://doi.org/10.2514/1.14006

[19] A. Altmikus, S. Wagner, P. Beaumier, and G. Servera, "A comparison: Weak versus strong modular coupling for trimmed aeroelastic rotor simulations," 06 2002.

[20] J. Abras, C. Lynch, and M. Smith, "Advances in rotorcraft simulations with unstructured CFD," *Annual Forum Proceedings - AHS International*, vol. 3, 01 2007.

[21] R. Marpu, L. Sankar, T. Norman, T. Egolf, and S. Makinen, "Analysis of the UH-60A rotor loads using wind tunnel data," 01 2013.

[22] R. Biedron and E. Lee-Rausch, "Computation of UH-60A airloads using CFD/CSD coupling on unstructured meshes," *American Helicopter Society 67th Annual Forum*, 05 2011.

[23] J. D. D. Boyd, "Hart-II acoustic predictions using a coupled CFD/CSD method," 2013.

[24] H. K. Lee, S.-H. Yoon, S. Shin, and C. Kim, "Coupled CFD/CSD analysis of a hovering rotor using high fidelity unsteady aerodynamics and a geometrically exact rotor blade analysis," 2008.

[25] B. Hübner, E. Walhorn, and D. Dinkler, "A monolithic approach to fluid-structure interaction using space-time elements," *Computer Methods in Applied Mechanics and Engineering*, vol. 193, p. 2087–2104, 06 2004.

[26] A. Boschitsch and T. Quackenbush, *Prediction of rotor aeroelastic response using a new coupling scheme*. [Online]. Available: https://arc.aiaa.org/doi/abs/10.2514/6.1994-2269

[27] J. A. Benek, J. L. Steger, F. C. Dougherty, and P. Buning, "Chimera. a grid-embedding technique," 1986.

[28] K. Lee, *3-D transonic flow computations using grid systems with block structure*. [Online]. Available: https://arc.aiaa.org/doi/abs/10.2514/6.1981-998

[29] P. Rubbert and K. Lee, *Numerical Grid Generation*, 1982, pp. 235–252.

[30] F. Lindner, M. Mehl, and B. Uekermann, *Radial Basis Function Interpolation for Black-Box Multi-Physics Simulations*, 05 2017.

[31] W. Press, S. A. Teukolsky, W. Vetterling, and B. P. Flannery, *Numerical Recipes*, 3rd ed. The Edinburgh Building, Cambridge CB2 8RU, UK: Cambridge University Press, 2007.

[32] "Technical documentation of the DLR TAU-code release 2018.1.0," Deutsches Zentrum für Luft- und Raumfahrt, Tech. Rep., 2018.

[33] D. Schwamborn, T. Gerhold, and R. Heinrich, "The DLR TAU-code: Recent applications in research and industry," 01 2006.

[34] T. Gerhold, M. Galle, O. Friedrich, and J. Evans, *Calculation of complex three-dimensional configurations employing the DLR-TAU-code*. AIAA ARC, 1997. [Online]. Available: https://arc.aiaa.org/doi/abs/10.2514/6.1997-167

[35] D. S. T. Gerhold, V. Hannemann, "On the validation of the dlr-tau code," in *New Results in Numerical and Experimental Fluid Mechanics II - Contributions to the 11th AG STAB/DGLR Symposium, Berlin, Germany, 1998*, W. Nitsche, H.-J. Heinemann, and R. Hilbig, Eds. Berlin, Heidelberg: Springer, 1999, pp. 426–433.

[36] C.-C. Rossow, N. Kroll, and D. Schwamborn, "The MEGAFLOW project - numerical flow simulation for aircraft," vol. 8, 01 2006.

[37] B. Eisfeld, "Implementation of reynolds stress models into the DLR-FLOWer code," 01 2004.

[38] W. Haase, B. Aupoix, U. Bunge, and D. Schwamborn, *FLOMANIA - a European initiative on flow physics modelling. Results of the European-Union funded project, 2002 - 2004*, 01 2006, vol. 94.

[39] W. Haase, M. Braza, and A. Revell, *DESider: A European effort on hybrid RANS-LES Modelling (Notes on numerical fluid mechanics and multidisciplinary design, Vol. 103)*, 01 2009, vol. 103.

[40] M. Galle, T. Gerhold, and J. Evans, *Parallel computation of turbulent flows around complex geometries on hybrid grids with the DLR-TAU code*, 12 2000, pp. 223–230.

[41] N. Kroll, T. Gerhold, S. Melber, R. Heinrich, T. Schwarz, and B. Schöning, *Parallel Large Scale Computations for Aerodynamic Aircraft Design with the German CFD System MEGAFLOW*, 01 2002, pp. 227–236.

[42] T. Alrutz, "Investigation of the parallel performance of the unstructured DLR-TAU-Code on distributed computing systems," 09 2005, pp. 509–516.

[43] A. A. Zaki, “Using tightly-coupled CFD/CSD simulation for rotorcraft stability analysis,” Ph.D. dissertation, Georgia Institute of Technology, 2012.

[44] W. Johnson, *CAMRAD II*. Palo Alto, California: Johnson Aeronautics, 2012, vol. 1.

[45] *TAU-Code User Guide.*

[46] G. Chourdakis, “A general OpenFOAM adapter for the coupling library preCICE,” Masterarbeit, Technical University of Munich, Oct 2017.

[47] B. Uekermann, “Partitioned fluid-structure interaction on massively parallel systems,” Dissertation, Institut für Informatik, Technische Universität München, Oct. 2016. [Online]. Available: https://mediatum.ub.tum.de/doc/1320661/document.pdf

[48] B. Wall, “2nd hhc aeroacoustic rotor test (hart ii) - part i: Test documentation -,” 01 2003.

[49] M. Smith, J. Lim, B. Wall, J. Baeder, R. Biedron, D. Jr, B. Jayaraman, S. Jung, and B.-Y. Min, “The HART II international workshop: an assessment of the state of the art in CFD/CSD prediction,” *CEAS Aeronautical Journal*, vol. 4, 12 2013.

[50] J. Lim and B. Wall, “Investigation in the effect of a multiple trailer free wake model for descending flights,” 01 2005, pp. TechnicalSession:DynamicsII–.

[51] A. Carnefix, “Analysis of low speed helicopter flight using a comprehensive rotorcraft model,” Technische Universität München, Tech. Rep., 2017.

[52] F. Simonis. (2019) preCICE config-visualizer. [Online]. Available: https://github.com/precice/config-visualizer

Appendices

A. User Guide

This guide will provide a short introduction to the different levels of customization required to start using the TAU and CAMRAD II preCICE adapters.

A.1 Getting Started with TAU

The TAU preCICE adapter can be integrated to an existing TAU simulation. There are three levels of customization possible using the TAU preCICE adapter, with more advance features requiring greater customization.

A.1.1 Level 1: Using pre-existing Solver and DataHandlers

There are two pre-existing cases which are implemented in this work:

1. A tightly-coupled simulation only passing force data from TAU to the solid solver and receiving displacement information on the shared interfaced.
2. A loosely-coupled isolated rotor-blade simulation passing force and moment data from TAU to the solid solver and receiving motion data in the form of Fourier series coefficients.

If only these two cases are required, it is possible that that the existing `TightSolver` and `HelicopterSolver` cases are suitable and no new code needs to be introduced.

Tightly-Coupled Simulation

The `TightSolver` should be applicable to any basic FSI simulation. The `TightSolver` employs a very generic TAU simulation loop. If your code requires any specific on-the-fly changes to the parafile or deviates significantly from the basic TAU simulation loop, please review the `tau_simulation_loop()` method found in `tight/tightsolver.py` in the TAU preCICE adapter root directory. This guide assumes that you are passing force data

along certain boundary markers and receiving displacement data on the same boundaries.

First, copy the `precice-config.xml`, `tau_config.yaml`, and `tau_coupled_script.py` from the Flap tutorial into your TAU simulation folder. Your folder structure should now look like Fig. A.1.

```
tausimulation
├── tau_coupled_script.py
├── precice-config.xml
├── tau_config.yaml
├── grid
│   ├── primary_grid.grid
│   └── ...
├── tau_script.py
├── tau_parafile
└── output
    ├── heli.pval.unsteady
    ├── boundary.bmap
    └── ...
```

Figure A.1: Basic TAU Simulation Folder Structure.

Then, update the `tau_config.yaml` file.

1. Fill in the logging location
2. Provide the TAU parafile and preCICE configuration file
3. Ensure solver type is TightSolver

Next, update the interfaces used in the simulation. There are two `DataHandler` subclasses created for the `TightSolver` case. `ForceHandler` writes TAU forces on the given boundary markers to the solid solver. `Displacement` handler reads displacements on the interface and writes a scatfile to the indicated location in the parafile. Your final `.yaml` file should look similar to Listing A.1.

The example shown in Listing A.1 gives an example of a coupled simulation with a single shared interface. There are two things the user must ensure for each interface. The name of each entry in `write_data` must start with "Forces" and the name of each entry in `read_data` must start with "Displacements" as the text is used to identify the type of `DataHandler` to instantiate. The next character after the `DataHandler` name must be a number or an underscore to

Listing A.1: Config Example

```
logging_location: ./log
preprocessing: false
participant: TAU
parafile: parafile_orig
precice_config: precice-config.xml
solver: TightSolver

interfaces:
- mesh: TAU_Mesh0
  boundary_markers: [5, 6, 15]
  write_data:
  - Forces0
  read_data:
  - Displacements0
```

prevent mistakes while parsing, i.e. `Forcesx` is not acceptable but "Forces_x" or "Forces0" is.

After filling in the `tau_config.yaml` check to ensure that the data types match the data names provided in the preCICE configuration file. Please refer to the preCICE wiki to learn how to fill this in, you may use the `precice-config.xml` in the Flap tutorial as a basis. Then, once these two have been set up, you may run ' python tau_coupled_script.py tau_config.yaml' to start up the simulation.

Loosely-Coupled Helicopter Simulation

There is greater difficulty in determining the `HelicopterSolver` is relevant to your simulation. The calculations when determining the forces and moment values are relatively involved. The specific handling of the data for in the Helicopter Solver tutorial is specific for coupling with CAMRAD II. This guide shows how to replace the helicopter simulation in the tutorial with a custom TAU simulation.

First, make a clone of the gitlab repository containing the `HelicopterSolver` class. Replace the parafile `parafile_orig` with your parafile. Ensure that the primary gridfile, boundary mapping file, motion file and output prefix for the TAU simulation are correct (they do not have to be in the same folder but relative paths from the folder where the script is executed must be correct). The adapter code also automatically tries to create folder locations if they do not exist.

Next, rename the yaml file to `tau_config_piecewise.yaml` or the `tau_config_full.yaml` file in the repository depending on the type of interpolation method to be used. Create the skeleton using the autogen scripts found in

"tutorial/config_autogen" in the Tutorials git repository.

1. Run the preCICE config generator, `precice_generator.py`. Run "python precice_generator.py -h" for more details.

2. Create a numpy object containing a list of the boundary markers for each blade in the simulation and order the blades in a clock-wise direction starting with the reference blade. For example, a simulation with 3 blades might contain $[[1, 3, 5], [7, 9, 11], [13, 15, 17]]$, with boundary markers $[1, 3, 5]$ relating to the reference blade

3. Run the TAU config generator, `tau_config_generator.py`. Run "python tau_config_generator.py -h" for more details.

These steps should create the working `precice-config.xml` and a skeleton for the TAU configuration yaml file. Next, update the configuration yaml file as follows.

1. Set the folder locations: `logging_location`, `logging_name`, and `restart_location`

2. Check that the default settings of the booleans `debug`, `precice`, `preprocessing`, `deformation_required` and `partitioning` are correct. Refer to Section B.3 for exact definitions. For the standard case, keeping the default values found in the git repository should be correct

3. Fill in the simulation details. Enter the number of azimuths to be passed per revolution. The number of blades in the simulation, the interpolation type, the revolutions per coupling (defaults to 2) and the number of initial revolutions.

The end result should look like Fig. A.2. Then, once these have been set up, you may run "python tau_coupled_script.py tau_config.yaml" to start up the simulation.

A.1.2 Level 2: Creating new DataHandlers and Config classes

The next level of customization is when a new `DataHandler` or `Config` subclass. This often comes about if a new data type needs to passed to the coupled simulation. In this case, simply subclass the respective base classes, described in Section C.1.1 and C.1.2.

Listing A.2: Sample config.yaml for TAU Simulation for HelicopterSolver

```yaml
logging_location: ./log
logging_name: tau_log

restart_location: ./restart

debug: true
precice: true
preprocessing: false
deformation_required: false
partitioning: true

participant: TAU
parafile: parafile_orig
precice_config: precice-config_piecewise.xml
solver: HelicopterSolver

simulation:
  total_azimuths: 24
  blades: 4
  interpolation_type: piece-wise
  revolutions: [1]
  initial_revolutions: 6
```

A.1.3 Level 3: Creating a new Solver class

The final level of customization is when a new `Solver` subclass is required. This comes about if the simulation loop logic needs to be changed. In this case, simply subclass the `Solver` class, as described in Section C.1.3

A.2 Getting Started with CAMRAD II

Unlike in TAU, where the TAU solver loop is modified via code, the CAMRAD II solver loop logic is modified by the input files. Hence, subclassing of the basic CAMRAD II `CamardSolver` class is not supported. The CAMRAD II preCICE adapter was designed to work specifically with the TAU preCICE adapter for loosely-coupled rotor blade simulations. Extending this adapter to be used in a more general way is a relatively involved process as modifying the code requires understanding of the structure of the coupled fluid solver (namely what data is passed from the fluid solver and in what format). The guide below demonstrates how to replace the CAMRAD II simulation in the given Helicopter tutorial. The second provides some guidelines to use the CAMRAD II preCICE adapter in a more general way.

A.2.1 Loosely-Coupled Helicopter Simulation

To replace the CAMRAD II simulation in the Helicopter tutorial, simply replace the inputs and the "Helicopter/CAMRAD" folder. Then, update the `camrad_config.yaml`.

1. Run the preCICE config generator, `camrad_config_generator.py`. Run "python camrad_config _generator.py -h" for more details.
2. Set the folder locations: `logging_location`, `logging_name`, `output_location`, `output_prefix`.
3. Set jobfile information: `jobfile` and `jobinfo_file`. `jobfile` is the script to run the CAMRAD II simulation. `jobfile_info` contains the rotor blade definitions
4. Set the preCICE configuration file: `precice_config`
5. Set if data is overwritten if it exists: `overwrite_data`
6. Fill in the simulation details:
   (a) `blades`: number of blades in the simulation
   (b) `blade_width` width of the blade (used to calculate intermediate grid)
   (c) `lengthwise_elements` and `widthwise_elements`: elements in each panel of intermediate grid
   (d) `delta_table_location` and `delta_table_prefix`: location of delta force tables

The final result should look something like Listing A.3.

Listing A.3: Sample config.yaml for CAMRAD II Simulation for Helicopter-Solver

```yaml
participant: CAMRAD
logging_location: ./log
logging_name: camrad_log
output_location: ./output
output_prefix: camout

job_file: ./jobs/job.lnx
job_info_file: ./input/HART_II_rotor.list

precice_config: ../TAU/precice-config_piecewise.xml

overwrite_data: false

simulation:
   results_dict: ./results/result
   blades: 4
   blade_width: 0.1
   lengthwise_elements: 10
   widthwise_elements: 5
   delta_table_location: ./tables
   delta_table_prefix: delta
```

B. Configuration File Definitions

This appendix describes the input files used to configure the coupled solution.

B.1 preCICE Configuration File

The preCICE configuration file is used to configure the preCICE coupling. An automated script used to generate a preCICE configuration file for loosely-coupled rotorcraft simulations has been created. This supports loosely-coupled rotorcraft using an serial-explicit coupling.

B.2 Parameters

B.3 TAU Configuration File

The TAU configuration file is stored as a yaml file. This file is produced when the `Config` class is dumped using the yaml library. This section only provides the basic variables required by the base `Config`, `Interface` and `Simulation` classes described in Section 5.4. This file is used to define the folder structure, choose the type of `Solver` subclass to be used in the simulation and link the various boundary markers in the boundary bmap file to preCICE data types. A script to automatically generate the boundary mapping links has been created for the loosely-coupled rotorcraft simulation.

Parameter Name	Type	Default	Required	Description
logging_location	str	N/A	True	Name of logging location
logging_name	str	N/A	True	Name of logging prefix
output_location	str	N/A	False	Name of output location, read from parafile directly

output_prefix	str	N/A	False	Name of output prefix, read from parafile directly
deform_location	str	N/A	False	Name of deformation location
deform_prefix	str	N/A	False	Name of deformation prefix
scatfile_location	str	N/A	False	Name of scatfile location
scatfile_prefix	str	N/A	False	Name of scatfile prefix
debug	bool	False	False	Set to True to enable debug output
precice	bool	True	False	Run with preCICE if set to True, else run standalone
preprocessing	bool	False	False	Runs TAU preprocessing if set to True
partitioning	bool	False	False	Repeats grid partitioning step if set to True, else reads partitioning from restart dict
deformation_required	bool	False	False	Set to true to enable deformation
participant	str	N/A	True	Name of TAU participant in precice xml config file. Must match with preCICE config to run
partner	str	N/A	False	Name of coupling partner to exchange variable data before precice interface initialization
shared_folder	str	N/A	False	Name of shared folder to be used to exchange variable data before preCICE interface initialization
parafile	str	N/A	True	Path to parafile location
precice_config	str	N/A	True	Path to preCICE xml config file
solver	str	N/A	True	Name of solver class used in coupled simulation

Table B.1: Parameters for TAU Config class

Parameter Name	Type	Default	Required	Description
name	str	N/A	False	Name of preCICE interface
mesh	str	N/A	True	name of preCICE mesh, must match with mesh name in preCICE config xml file
boundary_markers	list	N/A	False	list of TAU boundary markers related to this interface for mesh-based data
size	int	N/A	False	size of preCICE buffer for non mesh-based data
read_data	list	N/A	False	list of names of DataHandlers used to write to preCICE in coupled simulation. must match data name in preCICE config xml file
write_data	list	N/A	False	list of names of DataHandlers used to read from preCICE in coupled simulation. must match data name in preCICE config xml file

Table B.2: Parameters for TAU Interface class

Parameter Name	Type	Default	Required	Description
iteration	int	N/A	False	Number of coupled iterations. used when restarting at a certain coupled iteration

Table B.3: Parameters for TAU Simulation class

B.4 CAMRAD Configuration File

The CAMRAD II configuration file is stored as a yaml file. This file is produced when the `Config` class is dumped using the yaml library. This section only provides the basic variables required by the base `Config`, `Interface` and `Simulation` classes described in Section 5.5. This file is used to define the folder structure, choose the type of `Solver` subclass to be used in the simulation and link the various boundary markers in the boundary bmap file to preCICE data types. A script to automatically generate the boundary mapping links has been created for the loosely-coupled rotorcraft simulation.

B.5 Parameters

Parameter Name	Type	Default	Required	Description
logging_location	str	N/A	True	Name of logging location
logging_name	str	N/A	True	Name of logging prefix
output_location	str	N/A	False	Name of output location, read from parafile directly
output_prefix	str	N/A	False	Name of output prefix, read from parafile directly
deform_location	str	N/A	False	Name of deformation location
deform_prefix	str	N/A	False	Name of deformation prefix
scatfile_location	str	N/A	False	Name of scatfile location
scatfile_prefix	str	N/A	False	Name of scatfile prefix
debug	bool	False	False	Set to True to enable debug output
precice	bool	True	False	Run in coupled mode if set to True, else run standalone
overwrite_data	bool	True	False	Does not overwrite existing CAMRAD II output files if set to True
participant	str	N/A	True	Name of TAU participant in precice xml config file. Must match with preCICE config to run

partner	str	N/A	False	Name of coupling partner to exchange variable data before precice interface initialization
shared_folder	str	N/A	False	Name of shared folder to be used to exchange variable data before preCICE interface initialization
job_file	str	N/A	True	Path to CAMRAD II jobfile location
job_info_file	str	N/A	True	Path to CAMRAD II rotor definition location
precice_config	str	N/A	True	Path to preCICE xml config file
solver	str	N/A	True	Name of solver class used in coupled simulation

Table B.4: Parameters for CAMRAD II Config class

Parameter Name	Type	Default	Required	Description
name	str	N/A	False	Name of preCICE interface
mesh	str	N/A	True	name of preCICE mesh, must match with mesh name in preCICE config xml file
boundary_markers	list	N/A	False	list of TAU boundary markers related to this interface for mesh-based data
size	int	N/A	False	size of preCICE buffer for non mesh-based data
read_data	list	N/A	False	list of names of DataHandlers used to write to preCICE in coupled simulation. must match data name in preCICE config xml file
write_data	list	N/A	False	list of names of DataHandlers used to read from preCICE in coupled simulation. must match data name in preCICE config xml file

Table B.5: Parameters for CAMRAD II Interface class

Parameter Name	Type	Default	Required	Description
iteration	int	1	False	Number of coupled iterations. used when restarting at a certain coupled iteration
delta_table_location	str	N/A	True	location of output delta table
delta_table_prefix	str	N/A	True	prefix for output delta table
blades	int	N/A	True	number of blades for rotor blade simulation
azimuths	int	N/A	False	number of azimuth positions in one revolution of data
blade_width	float	N/A	True	width of rotor blade. used to construct intermediate grid for passing mesh-based data
blade_thickness	float	N/A	True	thickness of rotor blade. used to construct intermediate grid for passing mesh-based data
lengthwise_elements	int	N/A	False	number of lengthwise elements per panel in intermediate grid
widthwise_elements	int	N/A	True	number of widthwise elements per panel in intermediate grid
num_collocation_points	int	N/A	True	number of collocation points in simulation. will be filled by solver automatically
interpolation_type	float	N/A	False	type of interpolation (full | piecewise) received from fluid solver
num_harmonics	int	1	False	Number of coupled iterations. used when restarting at a certain coupled iteration
iteration	int	N/A	False	Number of coupled iterations. used when restarting at a certain coupled iteration

Table B.6: Parameters for CAMRAD II Simulation class

Parameter Name	Type	Default	Required	Description
radius	float	N/A	False	rotor blade radius as float
nblade	int	N/A	False	number of rotor blades in simulation as int
omega	float	N/A	False	rotational velocity of rotor blade in rad/s as float
ashaft	float	N/A	False	rotor shaft angle of attack in degrees as float
acant	float	N/A	False	rotor cant (roll) angle in degrees as float
rprop	list	N/A	False	radial stations of XQC and ZQC properties as list of floats
xqc	list	N/A	False	x-axis quarter chord offset from reference line in m as list of floats
zqc	list	N/A	False	z-axis quarter chord offset from reference line in m as list of floats
blade_spacing	list	N/A	False	azimuth angles (in degrees) of each blade in rotor as list of floats
redge	list	N/A	False	radial coordinates of blade panel edges as list of floats, length of number of panels + 1
rpos	list	N/A	False	radial coordinates of CAMRAD blade sensors as list of floats

Table B.7: Parameters for CAMRAD II RotorInfo class

C. User Extensibility

This chapter describes how the provided adapters can be extended. This guide is valid for the code submitted alongside this master's thesis. For the latest, up-to-date guide, please refer to the gitlab respository of the tauadapter and camradadapter.

C.1 Extending TAU Adapter

The TAU adapter can be extended in three ways: updating the input configuration, adding new datatypes to be passed, adding a new solver simulation loop or updating the input configuration. These three methods will be explored in this order

C.1.1 Updating Config class

Please refer to this guide if new input parameters are required during runtime. The adapter currently uses yaml files parsed using the `jsonobject` Python library. Hence, additional variables can be arbitrarily added to yaml files and referenced during runtime without modifying the classes. However, if additional type-checking is required or if a new parameter is necessary for running the simulation, one can subclass the `Config` class and add new parameters and input requirements. Please refer to the jsonobject PyPi guidelines for the exact datatypes supports by the `jsonobject` library: jsonobject.

It is highly recommended that users not modify the `Config` class directly as it should only contain the minimum inputs required to run a coupled TAU simulation. Adding additional requirements will introduce errors in other users code. Note that the `utility.configure()` does accept a Python class keyword argument which allows users to parse a given yaml file with a subclassed `Config` class at runtime. An example of a simple subclassing is shown in Listing C.1.

In this example, we replace the `interfaces` attribute in `Config` with a list

Listing C.1: Config class subclassing example

```
1  import jsonobject
2  from config import Config
3  from config import Interface
4
5  class MyInterface(Interface):
6          """ NewInterface includes additional data for rotorcraft simulation.
7
8          Attributes:
9              blade (int): blade number provided as int, required property
10             flight_speed (float): flight speed of rotorcraft as float, default
11         is 34.7
12         """
13
14         blade = jsonobject.IntegerProperty(required=True)
15         flight_speed = jsonobject.FloatProperty(default=34.7)
16
17
18 class MyDeformation(jsonobject.JsonObject):
19         """ MyDeformation stores additional data required for performing deformation
20
21         Attributes:
22             deformation_type (str): type of deformation performed, required
23
24         """
25         deformation_type = jsonobject.StringProperty(required=True)
26
27
28 class MyConfig(Config):
29         """ Customized Config class. Includes MyInterface and MyDeformation classes
30
31         Attributes:
32             interfaces (list): lis of MyInterface objects used to create
33             DataHandlers
34             deformation (MyDeformation): contains data used for deformation
35             deformation_required (bool): set to "true" if deformation required.
36             defaults to false.
37         """
38
39         interfaces = jsonobject.ListProperty(jsonobject.ObjectProperty(MyInterface))
40         deformation = jsonobject.ObjectProperty(MyDeformation)
41         output_file = jsonobject.StringProperty()
```

of `MyInterface`, defined in Line 5. This new class inherits from `Interface` but includes a new required attribute and provides a default `flight_speed` input.

We also create a new class `MyDeformation`, which inherits from the `JsonObject` base class determine the type of deformation used, defined in Line 18.

These two classes are now included in `MyConfig` along with a new string that stores the location of simulation output files.

Now that we have a new class used to parse out input file, we pass the specific class when before initializing the `Adapter` as seen in Listing C.2. The `Adapter` defaults to `Config` class if no new class is provided.

You may refer to the subclass of the `Config` class for the loosely-coupled he-

Listing C.2: Reading config file with MyConfig

```
from myconfig import MyConfig
from tauadapter.adapter import Adapter

...
adapter = Adapter(config_file)
adapter.update_config(MyConfig)
adapter.initialize()
adapter.setup_data_handlers()
adapter.execute()
...
```

licopter solver case, `HelicopterConfig`. In this example, two new jsonobject classes `Simulation` and `Deformation` were added to `Config`. Additionally, `HelicopterInterface` was created to provide azimuth position and blade number when setting up the `DataHandler` objects.

C.1.2 Extending DataHandler

Adding a new data type to be passed means extending the abstract base class `DataHandler`. The `DataHandler` class has the four virtual methods: `read()`, `write()`, `is_scalar()` and `get_dimension()`. These must be implemented by the new subclass in order to interface with the `DataInterface` class. The `DataHandler` base class has arguments `mesh_id`, `data_id`, `precice_vertex_ids` and `dimensions`. These are the preCICE mesh id, preCICE data id, the preCICE vertices as provided by the `set_mesh_vertices()` call and an integer of the dimensions of the simulation.

The method `is_scalar()` returns `True` if the handled data is scalar, otherwise it returns `False`. This determines the preCICE call made when passing or receiving data. The method `get_dimension()` returns the 2 or 3 depending if the simulation is run in 2D or 3D. These two methods should be implemented based on the data you intend to read or write.

The method `read()` will recieve a 1D numpy array of the given data type from preCICE and is intended to perform the post-processing of the data for TAU. The method `write()` should return a 1D numpy array of the data type to be written.

In the following example, we wish to write 3D velocity data to preCICE. We create a new class `VelocityHandler` that passes this data.

In this new class, we simply implement the 4 required virtual methods. The main work happens in Line 5, where the new read method should process and

Listing C.3: DataHandler class subclassing example

```
import jsonobject
from datahandler import DataHandler

class VelocityHandler(DataHandler):
        """ VelocityHandler passes 3D velocity ouput to preCICE
        """

    def __init__(self, args):
        """ Constructor for CollocationsHandler class.
        Arguments:
             args (dict): dictionary of multiple values, see DataHandler
                 base class for more information.
        """
        super(VelocityHandler, self).__init__(dict)

    def read(self, data, iteration):
        """ Not implemented!"""
        self.logger.error("We do not read Velocity data from Solid Solver!\n")
        raise NotImplementedError

    def write(self, iteration):
        """ Implements datahandlers.DataHandler.write() abstract function. Returns
        3D velocity data as a 1D array.
        Arguments:
            iteration (int): current coupled iteration as int
        Returns:
            returns velocity data as a 1D array of size a 1D array of size
            (3*self.length). data is arranged as [vx1, vy1, vz1, vx2, vy2, vz2 ...]
        """
        output_file = self.config.output_file
        # Read velocity data to variable 'data'
        return data

    def is_scalar(self):
        """Returns:
             Returns false as this is not a scalar data type
        """
        return False
```

output a 1D array that is passed to preCICE. Note that the base class `DataHandler` unpacks the `args` dict and provides several member attributes accessible to the user. The member attributes listed in Table C.1 are available in any `DataHandler` subclass.

Parameter Name	Description
self.tau_para	`TAU-Python` PyPara object used to update parafile
self.mesh_id	unique integer id identifying the specific mesh in preCICE interface
self.data_id	unique integer id identifying the specific data type in preCICE interface

self.length	number of nodes on interface (n) for current process as int
self.coordinates	1D numpy array of TAU-Code-Grid coordinates corresponding to preCICE vertex ids
self.precice_vertex_ids	1D numpy array of precice vertex ids
self.vertex_ids	1D numpy array of TAU vertex ids corresponding to precice vertex ids
self.interface	interface jsonobject (by default config.Interface class) used to store simulation input parameters
self.config	configuration jsonobject (by default config.Config class) used to store simulation input parameters
self.simulation	simulation jsonobject (by default config.Simulation class) used to store and pass simulation data between DataHandlers

Table C.1: Member Attributes of TAU DataHandler class

The new `DataHandler` can be passed to the The `Adapter` after initialization via a `dict`, as shown in Listing C.4.

Listing C.4: Reading config file with MyConfig

```
from myconfig import MyConfig
from mydatahandler import VectorHandler
from tauadapter.adapter import Adapter

...

DATA_READER = {"MyDataHandler": VectorHandler}

adapter = Adapter(config_file)
adapter.update_config(MyConfig)
adapter.initialize()
adapater.add_data_handlers(DATA_READER, "read")
adapter.setup_data_handlers()
adapter.execute()
...
```

The key for each new `DataHandler` ("MyDataHandler" in this case) is used to match `Interface` entries in the TAU config yaml file. An acceptable sample yaml file excerpt is shown in Listing C.5.

Listing C.5: Config

```yaml
...
interfaces:
- mesh: My_Vectors1
  read_data:
  - MyDataHandler1

- mesh: My_Vectors2
  read_data:
  - MyDataHandler_2

- mesh: My_Vectors3
  read_data:
  - MyDataHandler_III
...
```

All three data readers (MyDataHandler1, MyDataHandler_2 and MyDataHandler_III) will be recognized by the TAU adapter code as instances of the new `VectorHandler` class (see implementation in DataHandlerFactory class). Only letters (a-zA-z) are accepted as key values, numbers and underscore values are not acceptable key values. Recalling that the names of data readers must match their respective data type in the preCICE configuration xml file, this is to prevent issues from arising when enumerating multiple instances of the same reader (as is the case in the Listing). This also allows greater back-compatibility as existing preCICE adapters do not have a standardized method for enumerating datatypes in the preCICE configuration xml file.

You can refer to the `DataHandler` subclasses for the rotorcraft simulation, `MotionsHandler`, `AzimuthsHandler`, `CollocationsHandler`, `ForcesHandler` and `MomentsHandler` as examples of subclassing the `DataHandler` base class.

C.1.3 Extending Solver class

The `Solver` class can also be extended to introduce a new simulation loop. This particular subclassing is slightly more involved than introducing a new `DataHandler` or `Config` class. This is because some preCICE methods need to be included in the simulation loop. There are three methods that need to be considered when subclassing the solver class: the class constructor `__init__()`, the `execute()` method and the `execute_precice()` method. The subclassing process will be demonstrated via the following example.

We try to create a custom `Solver` class called `MySolver`. We begin by defining the class constructor. The `__init__()` method is shown in Listing C.6.

Listing C.6: Implementing constructor for `Solver` subclass

```
class MySolver(Solver):
    def __init__(self, config_file, data_interface):
        super(MySolver, self).__init__(config_file, data_interface)

        """
        self.data_interface = data_interface
        self.simulation = self.config.simulation
        self.precice_interface = data_interface.precice_interface
        """

        # Initialize tau_solver now to access node/vertex information for setting up
            data handlers
        self.logger.info("Initializing tau solver")
        self.tau_solver.init()
        tau_python.tau_parallel_sync()
```

Firstly, we need to ensure that the constructor arguments remain the same. The `Solver` class receives the name of the TAU config file as a str `config_file` and an instance of the `DataInterface` class, `data_interface`. Next, the base `Solver` should be instantiated via the call to `super()`. The `Solver` class instantiates the four `TAU-Python` classes PyPara, PyPrep, PySolv and PyDeform as `self.tau_para`, `self.tau_prep`, `self.tau_solver`, and `self.tau_deform` respectively. The base `Solver` class also instantiates the chosen `Config` class, filling the `self.config` member variable. The `PySolv` object should be initialized with the call to `init()` as seen in Line 14. This should be done on construction of the class to allow access to solver variables (such as inner iteration number or minimum residual) or TAU memory buffers. Otherwise, calls to these methods will cause TAU to crash. Of note is that the three commented lines between 5 - 9 are run in the base `Solver` class. You may store these variables under different names but please do not modify these member attributes. These are used when setting `DataHandler` objects and modifying these in the `Solver` subclass constructor may lead to unintended effects. Next, the `execute_precice()` method should be implemented. You simply need to include a few lines of preCICE code before you are done, see Listing C.7.

The important lines of code that must be included in every implementation are highlighted in blue. Firstly, the preCICE `Interface` object must be initialized, shown in Line 4. The initialization step is performed here because the setting up of `DataHandler` objects must occur before the preCICE `Interface` object can be initialized. Next, simulation loop control is handled by preCICE, shown in Line 9. Hence, loop condition is `is_coupling_`

Listing C.7: Implementing `execute_precice()` for `Solver` subclass

```
class MySolver(Solver):
    def execute_precice(self):

                self.precice_interface.initialize()

                if self.config.iteration is None:
                    self.config.iteration = 0

                while self.precice_interface.is_coupling_ongoing():
                        self.data_interface.initialize_data(self.config.iteration)

            if self.precice_interface.is_action_required(
                    precice.action_write_iteration_checkpoint()):
                # Write Checkpoint
                self.precice_interface.fulfilled_action(
                        precice.action_write_iteration_checkpoint())

             """ Insert TAU simulation code here """

            self.data_interface.write_all(self.coupling_iteration)
            if self.precice_interface.is_action_required(
                    precice.action_read_iteration_checkpoint()):
                    # Read Checkpoint
                self.precice_interface.fulfilled_action(
                        precice.action_read_iteration_checkpoint())
            else:
                    self.precice_interface.advance(1)
                    self.config.coupling_iteration += 1
            self.data_interface.read_all(self.coupling_iteration)

            """ Insert post-processing code here """

            tau_python.tau_parallel_sync()

        tau_python.tau_solver_output_field()
        tau_python.tau_solver_output_surface()
        tau_python.tau_parallel_sync()
        self.tau_solver.stop()
        self.precice_interface.finalize()
```

`ongoing()`, which returns True as long the preCICE coupling runs. Then next line is analogous to the preCICE `initialize_data()` call, which reads data from other participants if a serial coupling scheme is used (refer to Section 4.4.2 for more details).

The actual TAU simulation loop should be inserted at Line 18. For a tightly-coupled simulation, this should just be single outer loop iteration. For a loosely-coupled simulation, another loop should be used which runs the simulation for the period between coupling steps.

After the TAU simulation loop is run, TAU should be in a state ready for the coupling step. The `DataInterface` class wraps the complicated reading and write of this step. Simply write data to other coupling participants (Line 20, advance time in the coupled simulation (Line 28 and read data from the new timestep (Line 29).

At this point, the code in lines 12 - 16 and 21 -25 have not been discussed. These code segments are only used if an implicit coupling scheme is used. This requires the saving and reading of checkpoints. This is not discussed in this work because only explicit coupling schemes are used. However, this syntax may be helpful if you with to do so as an extension of this work.

Now, the new `Solver` subclass can be passed to the `Adapter`, as shown in Listing C.8.

Listing C.8: Reading config file with MyConfig

```
1  from myconfig import MyConfig
2  from mydatahandler import VectorHandler
3  from mysolver import MySolver
4  from tauadapter.adapter import Adapter
5
6  ...
7
8  DATA_READER = {"MyDataHandler": VectorHandler}
9
10 adapter = Adapter(config_file)
11 adapter.update_config(MyConfig)
12 adapter.update_solver(MySolver)
13 adapter.initialize()
14 adapater.add_data_handlers(DATA_READER, "read")
15 adapter.setup_data_handlers()
16 adapter.execute()
17 ...
```

The method is similar to that for updating the `Config` class, simply call `update_solver()` before calling `initialize()`.

C.2 Extending CAMRAD II Adapter

The CAMRAD II adapter should require less modification compared to the TAU adapter. This is because changes to the simulation loop is handled by input files. Instead, only the `Config` and the `DataHandler` class can be customized.

C.2.1 Updating Config class

Updating the `Config` class for the CAMRAD II adapter is identical to that when updating the TAU adapter. Simply subclass the base `Config` class using the same steps in Section C.1.1. The code to update the `Config` class is also identical, shown in Listing C.9.

Listing C.9: Reading config file with MyConfig

```
1 from myconfig import MyCamradConfig
2 from camradadapter.adapter import Adapter
3
4 ...
5 adapter = Adapter(config_file)
6 adapter.update_config(MyCamradConfig)
7 adapter.initialize()
8 adapter.execute()
9 ...
```

C.2.2 Update DataHandler class

Updating the `DataHandler` class is different to that in TAU. As CAMRAD II has fewer pieces of information required to perform postprocessing, the amount of information passed to the base `DataHandler` class is also reduced. This allows us to clearly pass each required argument to the `DataHandler` without use of a dict. An example of an extended CAMRAD II `DataHandler` class is below.

As the CAMRAD II outputs are read from ASCII files, it is easiest to find data via use of regex. It may be helpful to refer to the existing `DataHandler` subclasses for CAMRAD II as a base and replacing the regex with what is required or adding additional post-processing logic to get what is desired.

Listing C.10: DataHandler class subclassing example for CAMRAD II

```
import jsonobject
from datahandler import DataHandler

class MotionHandler(DataHandler):
        """ MotionHandler passes 3D motion ouput to preCICE
        """

    def __init__(self, mesh_id, data_id, precice_vertex_ids):
        """ Constructor for CollocationsHandler class.
        Arguments:
             args (dict): dictionary of multiple values, see DataHandler
                 base class for more information.
        """
        super(MotionHandler, self).__init__(mesh_id, data_id, precice_vertex_ids)

        self.mesh_id = mesh_id
        self.data_id = data_id
        self.precice_vertex_ids = precice_vertex_ids
        # Number of dimensions in problem
        self.length = len(self.precice_vertex_ids)


    def read(self, data, iteration):
        """ Not implemented!"""
        self.logger.error("We do not read Velocity data from Solid Solver!\n")
        raise NotImplementedError

    def write(self, iteration):
        """ Implements datahandlers.DataHandler.write() abstract function. Returns
        3D velocity data as a 1D array.
        Arguments:
            iteration (int): current coupled iteration as int
        Returns:
            returns motion data as a 1D array of size a 1D array of size
            (3*self.length). data is arranged as [x1, y1, z1, x2, y2, z2 ...]
        """
        output_file = self.config.output_file
        return data

    def is_scalar(self):
        """Returns:
             Returns false as this is not a scalar data type
        """
        return False
```

D. Recreation of Data

This guide will provide documentation on the exact input parameters to recreate all results presented in this work.

D.1 Toy Example: Perpendicular Flap

The results for the Toy Example was run on the cluster at the Chair for Helicopter Technologies using the code present in the TAUadapter git repository [1] with commit SHA `cccdae1d` and tagged as “Flap_Case_Reference”.

D.2 Case 1: Loose Coupling without Deformation

D.2.1 Coupling Results

The coupling results for Case 1 presented in this work were run on the cluster at the Chair for Helicopter Technologies using the code present in the TAU adapter git repository Footnote 1 with commit SHA `37164a60` (tagged as “Case_1_Reference”) and the CAMRAD II adapter git repository [2] with commit SHA `e287d26e` (tagged as “Case_1_Reference”). The tutorial code can be found in the `Tutorials/Helicopter` folder.

Seven initial revolutions were run in TAU before this coupling was run. These initial revolutions were run on the LRZ with a specific motion and parafile (both passed to the Chair of Helicopter Technologies) and available in a reference gitlab repository [3] under the `Helicopter/TAU` folder. The bash script used to run the simulation on the LRZ is also provided.

D.2.2 Profiling Results

The profiling of the Case 1 code (same commits as above) against the baseline case was run on the cluster at the Chair for Helicopter Technologies on node

[1] https://gitlab.lrz.de/KeefeHuang/tauadapter [2] https://gitlab.lrz.de/KeefeHuang/camradadapter
[3] https://gitlab.lrz.de/KeefeHuang/adapter-results-reference

TUM-WE140. The baseline code is from the Chair of Helicopter Technologies. Several changes were made to the simulation settings to compare the coupling:

1. Number of inner iterations was changed from 250 to 0
2. Number of coupling steps was changed to 2 (preCICE only runs for 2 coupling iterations)

The exact files used to run the profiling case can be found in the reference repo Footnote 3 under the `Profile` folder. The bash script used to run on the Chair's cluster (`sbatch.cmd`) and the output cprofiles for the TAU and CAMRAD II script (`tau.cprofile` and `camrad.cprofile`) are also provided. The output from the original coupling is provided as `coupled.cprofile`.

D.3 Case 2: Loose Coupling with Deformation

The deformation results for Case 2 were run on the LRZ using the code present in the TAU adapter git repository Footnote 1 with commit SHA `37164a60` (tagged as "Case_1_Reference") and the CAMRAD II adapter git repository Footnote 2 with commit SHA `e287d26e` (tagged as "Case_1_Reference"). The tutorial code can be found in the `Tutorials/Deformation` folder. The exact CAMRAD II and TAU input files can be found in the reference repo Footnote 3 under the `Deformation` folder. The bash script used to run the simulation on the LRZ is also provided. The simulation was run for one coupling iterations and the data from CAMRAD II output "camout1" and TAU azimuths 1080 to 1440 were analyzed.